UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

TESTING ACTIVO Y PASIVO DE SISTEMAS CON INFORMACIÓN TEMPORAL Y PROBABILÍSTICA.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

César Andrés Sánchez

Bajo la dirección de los doctores

Mercedes García Merayo
Manuel Núñez García

Madrid, 2010

ISBN: 978-84-693-6327-0

Testing Activo y Pasivo de Sistemas con Información Temporal y Probabilística

TESIS DOCTORAL

César Andrés Sánchez
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Directores: Mercedes García Merayo y Manuel Núñez García

Publicaciones

Esta tesis doctoral se presenta en *formato publicaciones*, de acuerdo con el apartado 4.4 del acuerdo del Consejo de Gobierno de fecha 14 de octubre de 2008, en el que se aprueba la normativa de Desarrollo del Régimen relativo a elaboración, tribunal, defensa y evaluación de la Tesis Doctoral del Real Decreto 1393/2007, de 29 de octubre (BOE de 30 de octubre), por el que se establece la ordenación de las enseñanzas universitarias ofciales de la Universidad Complutense de Madrid. Dichas publicaciones recogen todos los resultados que han sido obtenidos en los diferentes trabajos de investigación desarrollados con el fin de alcanzar el objetivo fijado para la realización de la tesis. A continuación se relacionan los artículos que integran la tesis agrupados en cuatro bloques, teniendo en cuenta sus diferentes contenidos temáticos: Testing Activo de Sistemas con Información Probabilística, Testing Pasivo de Sistemas con Información Temporal, Aplicación de Técnicas de Testing Activo en Testing Pasivo y Casos de Estudio.

I) Testing Activo de Sistemas con Información Probabilística

a) C. Andrés, L. Llana, and I. Rodríguez. Formally comparing user and implementer model-based testing methods. In *4th Workshop on Advances in Model Based Testing, A-MOST'08*, pages 1–10. IEEE Computer Society Press, 2008.

b) C. Andrés, L. Llana, and I. Rodríguez. Formally transforming user-model testing problems into implementer-model testing problems and viceversa. *Journal of Logic and Algebraic Programming*, 78(6):425–453, 2009.

II) Testing Pasivo de Sistemas con Información Temporal

a) C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of timed systems. In *6th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'08, LNCS 5311*, pages 418–427. Springer, 2008.

b) C. Andrés, M.G. Merayo, and M. Núñez. Formal correctness of a passive testing approach for timed systems. In *5th Workshop on Advances in Model Based Testing, A-MOST'09*, pages 67–76. IEEE Computer Society Press, 2009.

c) C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of stochastic timed systems. In *2nd Int. Conf. on Software Testing, Verification, and Validation, ICST'09*, pages 71–80. IEEE Computer Society Press, 2009.

d) C. Andrés, M.G. Merayo, and C. Molinero. Advantages of mutation in passive testing: An empirical study. In *4th Workshop on Mutation Analysis, Mutation'09*, pages 230–239. IEEE Computer Society Press, 2009.

III) Aplicación de Técnicas de Testing Activo en Testing Pasivo

a) C. Andrés, M.G. Merayo, and M. Núñez. Using a mining frequency patterns model to automate passive testing of real-time systems. In *21st Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'09*, pages 426–431. Knowledge Systems Institute, 2009.

b) C. Andrés, M.G. Merayo, and M.Ñúñez. Supporting the extraction of timed properties for passive testing by using probabilistic user models. In *9th Int. Conf. on Quality Software, QSIC'09*, pages 145–154. IEEE Computer Society Press, 2009.

IV) Casos de Estudio

a) C. Andrés, S. Maag, A. Cavalli, M.G. Merayo, and M. Núñez. Analysis of the OLSR protocol by using formal passive testing. In *16th Asia-Pacific Software Engineering, APSEC'09*, pages 152–159. IEEE Computer Society Press, 2009.

b) C. Andrés, M.G. Merayo, and M. Núñez. Formal passive testing of timed systems: A case study with the Stream Control Transmission Protocol. In *7th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM'09*, pages 73–82. IEEE Computer Society Press, 2009.

Resumen

Determinar la *existencia o ausencia* de fallos dentro de los actuales sistemas informáticos es una tarea necesaria para incrementar su fiabilidad. La técnica más usada en entornos industriales para detectar fallos en los sistemas se denomina *testing*. A diferencia del testing clásico, que se efectúa manualmente, las empresas cada vez buscan más la forma de automatizar la fase de pruebas, así como el estudio de propiedades no triviales para la adecuación y mejora del producto. Por ello, hoy en día se va imponiendo más el llamado *testing formal*, donde se aplican los *métodos formales* dentro de la metodología de testing. Los métodos formales son técnicas matemáticas que tratan la construcción y/o el análisis de modelos matemáticos que contribuyen a la automatización del desarrollo de sistemas informáticos fiables. Los métodos formales se suelen utilizar en aquellos entornos donde las condiciones de seguridad y de criticidad son altas. Por ejemplo, sistemas donde se ponen vidas humanas en manos de máquinas, ya sea en sistemas de salud, medios de locomoción, sistemas de investigación biológica, etc. En esta combinación de testing con métodos formales, los métodos formales proporcionan rigor matemático, automatización, modelado formal y capacidad para el estudio de propiedades, tanto a nivel de especificación como a nivel de construcción de los tests.

Dentro del testing existen muchos paradigmas y tipologías. En esta tesis nos centramos en una clasificación basada en *cómo* se aplican los tests en el sistema bajo testeo. Las técnicas bajo este criterio se clasifican en técnicas de *testing activo* y de *testing pasivo*. En el testing activo, un testeador tiene total libertad para probar cualquier batería de tests. La existencia o ausencia de errores tras la aplicación de estos tests ayudará al experto a dar un veredicto sobre la fiabilidad del producto. Por otro lado, en el testing pasivo no se permite la aplicación

directa de pruebas al sistema. En este caso, la comprobación de la existencia o no de errores se realiza "observando" cómo se ejecuta el programa o viendo el histórico de ejecución del sistema.

En esta tesis trataremos estos dos tipo de testing. En primer lugar, presentamos una metodología de testing activo de sistemas con información probabilística. A continuación, nos centramos en metodologías de testing pasivo para sistemas con información temporal. Finalmente, combinamos estas dos aproximaciones en un marco de testing pasivo donde la extracción de invariantes se realiza aplicando los modelos probabilísticos de usuario desarrollados en la primera parte de la tesis.

Para concluir este resumen, cabe destacar que una de las críticas que siempre aparece a la hora de utilizar métodos formales, y en particular a la hora de utilizar modelos matemáticos para el cómputo de ciertas propiedades, es la ausencia de casos de estudio reales. En esta tesis hemos querido mostrar que la utilización de las técnicas que hemos estudiado permite una buena integración con el mundo industrial, y con herramientas que se utilizan en la actualidad en entornos no académicos.

Agradecimientos

Mi más profundo agradecimiento a mis directores Mercedes García Merayo y Manuel Núñez García por su ayuda, así como por la confianza que depositaron en mí. También quiero agradecer a los revisores y miembros del tribunal de esta Tesis. Mención especial merece David de Frutos por su cuidadosa lectura. Es difícil echar la vista atrás y sintetizar en unos pocos nombres toda la gente que me ha dado su apoyo, con la que he estado compartiendo momentos inolvidables, momentos caóticos, momentos para el recuerdo y momentos para el olvido. Sin esta interacción, yo no sería como soy, ni estaría escribiendo estas líneas para todos vosotros.

Gracias.

El momento más peligroso viene tras la victoria.

Napoléon Bonaparte

Many people have helped me in various ways during my struggle for being what I am. It is impossible to mention and thank them all. My deepest gratefulness to everybody that has helped me to write this Thesis, in particular, to my advisors Mercedes García Merayo and Manuel Núñez García for their help and support, as well as for the trust they put on me. I would also like to thank the reviewers and Committee members of this Thesis. In particular, I would like to thank David de Frutos for the careful reading of a draft version of this PhD Thesis.

Thanks.

The most dangerous moment comes with victory.

Napoléon Bonaparte

ÍNDICE GENERAL

CAPÍTULO 1

Introducción

Nadie se nos montará encima si no doblamos la espalda.

Martin Luther King

Los *métodos formales* hacen referencia a técnicas matemáticas para la especificación, desarrollo y verificación de sistemas software y hardware. La aplicación de los métodos formales es especialmente relevante en sistemas en los que, debido a razones de seguridad, es necesario establecer que durante el proceso de desarrollo no se han producido errores. La representación formal de los sistemas permite un análisis riguroso de sus propiedades. En particular, permite establecer la corrección del sistema final respecto a la especificación, el cumplimiento de las condiciones requeridas para el mismo, la equivalencia semántica con otros sistemas, el nivel de preferencia de un sistema respecto a otro en base a un determinado criterio, la existencia de posibles comportamientos incorrectos, etc.

En lo referente a las técnicas formales para el establecimiento de la corrección de un sistema respecto a una especificación, las metodologías de *testing formal* han jugado un papel muy relevante. La aplicación de dichos métodos requiere la identificación de los aspectos críticos del sistema, esto es, aquellos aspectos que marcan la diferencia entre lo correcto y lo incorrecto. Inicialmente, las metodologías de testing estaban enfocadas al análisis del comportamiento funcional de los sistemas, es decir, a determinar, por una parte, si el sistema testeado realizaba las acciones requeridas, y por otra a garantizar que no realizaba las acciones prohibidas. Sin embargo, en el caso de los sistemas de tiempo real adquieren tanta relevancia los aspectos cuantitativos como los cualitativos: es tan importante verificar que los sistemas *hacen lo que deben hacer*, como que *lo hacen cuándo y cómo deben hacerlo*. Debido a ello, durante los últimos años, las técnicas de testing formal se han orientado también hacia el tratamiento de las propiedades no funcionales de los sistemas, ya que estos aspectos pueden ser críticos, como por ejemplo la probabilidad de que se realice una acción, o el tiempo consumido por el sistema para realizarla.

Los principales objetivos de esta tesis son, por un lado el estudio y extensión de modelos formales con información probabilística para la realización de testing activo, y por otro proveer de una metodología para la realización de testing pasivo en sistemas con información temporal. En lo concerniente a la descripción de propiedades probabilísticas, los nuevos modelos pueden expresar la probabilidad que tienen los usuarios del sistema de realizar una determinada acción, mediante el modelo de usuario; o la probabilidad de que ocurra un fallo durante la implementación del sistema, mediante el modelo de implementador. En lo concerniente a la inclusión de tiempo, se consideran diferentes formalismos, tomando como punto de partida las máquinas de estados finitos, para modelar sistemas donde el tiempo se denota de distintas maneras, y se define una vía de representación fácil y viable de propiedades con información funcional y no funcional, para testearlos.

Por otro lado destacamos, como otro objetivo de la tesis, la integración de las metodologías de testing formal estudiadas fuera del entorno académico. En simbiosis con el marco teórico, en nuestro grupo de investigación hemos desarrollado una herramienta donde están implementadas todas las funcionalidades del mismo. Durante el desarrollo de esta herramienta se ha trabajado tanto en que su funcionamiento sea correcto, como en que su diseño sea modular, lo que facilita las tareas de adaptación, ampliación y mantenimiento. Se pretende que todo esto converja en una herramienta atractiva no sólo de manera individual, sino como módulo integrado en otros proyectos software.

La presente tesis doctoral se presenta en *formato publicaciones*, de acuerdo con el apartado 4.4 del acuerdo del Consejo de Gobierno de fecha 14 de octubre de 2008, en el que se aprueba la normativa de Desarrollo del Régimen relativo a elaboración, tribunal, defensa y evaluación de la Tesis Doctoral del Real Decreto 1393/2007, de 29 de octubre (BOE de 30 de octubre), por el que se establece la ordenación de las enseñanzas universitarias oficiales de la Universidad Complutense de Madrid. En dicho artículo se contempla que se podrán presentar tesis doctorales en formato publicaciones entendiéndose como tal las tesis constituidas básicamente por publicaciones editadas o aceptadas para su publicación. Dichas publicaciones recogen los resultados que han sido obtenidos en diferentes trabajos de investigación desarrollados con el fin de alcanzar el objetivo fijado para la realización de la tesis. En nuestro caso todos ellos han sido refrendados con su publicación en diferentes congresos y revistas de ámbito internacional, contando por tanto con la subsiguiente *"evaluación inter pares"*, y habiendo sido ratificados tras los correspondientes rigurosos procesos de selección.

Los artículos que integran esta tesis doctoral se presentan agrupados en cuatro grandes bloques, teniendo en cuenta sus diferentes contenidos temáticos: testing activo de sistemas con información probabilística, testing pasivo con información temporal, aplicación de las técnicas de testing activo con información probabilística en el testing pasivo, y aplicabilidad de estas técnicas en casos de uso reales, así como la integración en herramientas no académicas. Aunque a primera vista las dos primeras técnicas pueden parecer bastante diferentes, la tercera parte de esta tesis consigue una simbiosis entre ambas, permitiendo el trasvase de los resultados contrastados de una metodología a la otra.

1.1. Discusión integradora y objetivos

En esta sección se presenta una *discusión integradora* sobre las publicaciones que constituyen esta tesis. El *testing* es una técnica que se utiliza para la detección de errores en sistemas tanto software como hardware. El testing puede hacer uso de métodos formales a la

hora de generar de forma semi-automática conjuntos de tests, estando el estudio de los mismos bien asentado [PYB96, LY96, CFP96, Mil98, MA00, MA01, BT01, RMN08, HBB+09]. En las metodologías de testing, los requisitos de un programa vienen habitualmente definidos por un modelo (especificación) que denota qué debería hacer el programa (requisitos funcionales) o cómo debería hacerlo (requisitos no funcionales).

Entre las diferentes clasificaciones existentes de las metodologías de testing, en esta tesis, nos centramos en la que considera *cómo* se interactúa con el sistema que se ha de testear. Esta clasificación establece dos categorías: testing *activo* y testing *pasivo*. En testing activo el testeador tiene acceso directo al sistema, es decir, puede aplicar una batería de tests definida previamente y observar los resultados. Por otro lado, el testing pasivo se basa en la idea de que el testeador no tiene acceso directo al sistema. Esta situación puede darse, por ejemplo, en grandes bases de datos donde no es posible la interacción directa entre testeador y sistema, es decir, no puede añadir nuevos datos, borrarlos, modificar la estructura, etc. En este caso, se tiene que determinar si el comportamiento del sistema es correcto o no mediante la observación de las interacciones realizas por el sistema, comprobando si éstas son correctas respecto a un *conjunto de propiedades* extraído de la especificación. El testing pasivo, aunque con otras denominaciones, es una técnica que aparece documentada en la literatura al menos desde finales de los años setenta [AAD79]. Durante los últimos 15 años han aparecido varias propuestas para realizar testing pasivo con la novedad de incluir una componente formal [LNS+97, TC99, LCH+02, ACN03, ACC+04, BCNZ05, LCH+06, UX07, BDS+07]. En esta línea, [BCNZ05] representa una forma novedosa de afrontar el testing pasivo. En esta metodología, las propiedades que se desean testear se representan mediante una serie de *invariantes*. Esta aproximación será la que consideremos como paso inicial en esta tesis a la hora de realizar testing pasivo de sistemas con información temporal.

La generación de tests (respectivamente propiedades), que se utilizan en testing activo (respectivamente pasivo), se puede realizar de diferentes formas. Una de ellas consiste en razonar sobre el conjunto total de tests que se puede generar, seleccionando algunos de ellos de acuerdo con algún criterio heurístico; suele ser habitual extraer en primer lugar aquellos tests que tengan una mayor probabilidad de detectar errores en el programa. Otra posibilidad es utilizar modelos probabilísticos para la generación de estos tests. El estudio de modelos probabilísticos se remonta a principios de los años sesenta [Rab63, Paz71], pero es en los últimos años cuando se han empezado a tener en cuenta los aspectos probabilísticos a la hora de testear formalmente sistemas [LNR06c, CSV07]. Una línea de trabajo novedosa apareció en [LNR06a, LNR06b] donde las probabilidades se consideraban no sólo en la forma

clásica, es decir, integrada en los tests, en las especificaciones, y/o en las implementaciones, sino que se añadían otros *actores*, como es el caso de los usuarios de los sistemas. Estas dos aportaciones son el punto de partida para tratar información probabilística en la metodología de testing activo con modelos de usuario y de implementador, propuesta en esta tesis.

La primera parte de esta la tesis engloba las contribuciones: *Formally comparing user and implementer model-based testing methods* [ALR08], y *Formally transforming user-model testing problems into implementer-model testing problems and viceversa* [ALR09]. Estas publicaciones tienen como objetivo la investigación de métodos formales en el testing activo de sistemas, el modelado de ciertas propiedades, y la introducción de sistemas con información probabilística para la generación de tests.

Normalmente el principal inconveniente al testear cualquier sistema es, por un lado, el elevado número de tests que hay que aplicar y, por otro lado, saber que la corrección absoluta de un programa no se puede asegurar, en general, mediante testing. Teniendo en cuenta ambos factores, en [ALR08] se presenta una metodología para generar un número finito de tests *relevantes*, a partir de una especificación formal utilizando modelos probabilísticos. Se considera que un conjunto de tests relevantes debe contener aquellos tests que tienen un grado de representación importante.

En [ALR08] se estudian dos modelos probabilísticos diferentes, los cuales son una pieza clave a la hora de extraer una batería de tests relevante. El primero de ellos sirve para definir *modelos de usuario*. Un modelo de usuario es una máquina de estados probabilística donde está plasmada la probabilidad que tienen los distintos usuarios de interaccionar de manera diferente con el sistema. Supongamos que tenemos un cajero automático. Este cajero lo tenemos en fase de pruebas y guardamos todas las operaciones de los usuarios: el usuario obtiene dinero, el usuario ve sus últimos movimientos de cuenta, el usuario cancela la operación, etc. Teniendo un número relevante de datos dentro de ese registro, seremos capaces de expresar el comportamiento de los usuarios de la siguiente manera: "El 50 % de los usuarios que han interactuado con el sistema han sacado dinero", "el 1 % de los usuarios ha pulsado el botón cancelar", etc. Todo este tipo de información será incluida en el modelo de usuario.

El segundo modelo probabilístico estudiado es un *modelo de implementador* del sistema. En él, el valor asociado con cada transición representa la probabilidad de que se haya producido un error a la hora de implementar el sistema. Este modelo sirve para representar situaciones típicas como la utilización de variables compartidas, donde se representa la probabilidad de generar un fallo en la lectura, escritura, actualización, o para expresar que "siempre que un implementador falla en la primera transición, con una expectativa de un

80 % fallará la siguiente".

Después de definir ambos modelos probabilísticos se proporciona la noción de cobertura de un conjunto de tests. En el modelo de usuario ésta viene ligada al *grado de representatividad* asociado a dicho conjunto. Por ejemplo, volviendo al ejemplo anterior, si tenemos que el 50 % de los usuarios que han interactuado con el sistema han realizado un reintegro, y nuestro conjunto de tests se centra en esta funcionalidad, entonces podemos decir que se ha cubierto el 50 % de todas las posibles interacciones de los usuarios con el sistema, es decir la cobertura de ese conjunto de tests sería del 50 %. En el modelo de implementador, la cobertura del conjunto de tests denota la capacidad de *encontrar un fallo* en la implementación tras aplicarlo.

En este punto disponemos de dos modelos probabilísticos. Cada uno de ellos corresponde a diferentes informaciones sobre el funcionamiento del sistema y cuentan además con unas nociones de cobertura diferentes. En [ALR09] demostramos, que dado un modelo de usuario U podemos tener un conjunto finito de tests óptimos para chequear el mismo y, que dado un modelo de implementador I podemos tener un conjunto de tests óptimos sobre él. Con frecuencia podemos encontrarnos con una situación en la que tenemos un determinado modelo construido y, para aplicar la metodología de testing necesitamos el otro, ya que la extracción de un conjunto óptimo de tests sobre éste puede ser más fácil. En los casos en los que sólo tenemos la información suficiente para hacer un determinado tipo de modelo, mostramos cómo obtener de manera automática un modelo probabilístico equivalente sin pérdida de información. Dicho de otro modo, si tenemos información suficiente para crear el modelo de usuario U, pero a la hora de obtener el conjunto óptimo de tests necesitamos un modelo de implementador, entonces se puede crear un modelo de implementador $I_U = f(U)$ y generar el conjunto óptimo de tests para I_U. En [ALR09] también se demuestra que el conjunto de tests óptimos para U sería el mismo que el conjunto óptimo para I_U. El paso del modelo de implementador al modelo de usuario también está automatizado, y la optimalidad de los conjuntos de tests también está asegurada.

La segunda parte de la tesis es una continuación de la investigación iniciada en el *Máster de Investigación en Informática* durante el curso académico 2007/2008, aportando contribuciones originales en la línea de *extensión temporal de testing pasivo mediante invariantes* [And08]. Los artículos que se engloban dentro de este grupo son: *Passive testing of timed systems* [AMN08], *Formal correctness of a passive testing approach for timed systems* [AMN09a], *Passive testing of stochastic timed systems* [AMN09c], y *Advantages of mutation in passive testing: An empirical study* [AMM09].

La metodología que se toma como punto de partida en esta parte de la tesis fue presentada en [CGP03, ACN03, BCNZ05] y está basada en el concepto de *invariantes*. Los invariantes denotan las propiedades más importantes que se deben cumplir en cualquier estado de la máquina bajo estudio. Dado que en esta parte de la tesis se trabaja con información temporal, uno de los principales puntos que tenemos que tener en cuenta en nuestro marco teórico es la representación del tiempo. Hemos optado por usar dos modelos distintos. Mientras que en [AMN08, AMN09a] se utilizan tiempos fijos, en [AMN09c, AMM09] el tiempo se representa mediante variables estocásticas.

Nuestra primera aproximación al testing pasivo de sistemas con información temporal culminó con [AMN08]. En este trabajo se parte de una especificación formal que está representada mediante una extensión de las máquinas de estados finitos, para poder incluir información temporal, y de un sistema a testear con el que no se puede interactuar. Ante esta situación, lo primero que tenemos que hacer es grabar las interacciones del sistema en un *log*, para después analizarlo. Dado que el objetivo de este primer trabajo es proporcionar un marco teórico a la aplicación de testing pasivo de sistemas con información temporal, además de definir formalmente el modelo para representar especificaciones, los logs del sistema, y la sintaxis para expresar invariantes, se proporcionan algoritmos para comprobar que los logs no violan los requerimientos impuestos por los invariantes.

Un punto que no hemos discutido todavía es cómo se obtienen los invariantes y cómo podemos asegurarnos de que los invariantes no están en contradicción con la especificación. Existen varias formas de obtener dichos invariantes. La utilizada en este artículo considera que el conjunto de invariantes es propuesto por los testeadores. Si bien suponemos que se trata de expertos, éstos no tienen porqué conocer perfectamente la especificación, por lo que una vez definidos los invariantes, se debe comprobar su corrección con respecto a la especificación. En [AMN08] se presenta un algoritmo para determinar dicha corrección.

En este trabajo también se incluye la primera versión de nuestra herramienta: PASsive TEsting (PasTe)[1]. Esta herramienta ha sido ampliada con el paso del tiempo. La primera versión estaba implementada en Java, estando constituida por cuatro módulos. El primero de ellos permite especificar los sistemas, proporcionando métodos para introducir modelos formales con información temporal dentro del sistema. El segundo se encarga del tratamiento de los logs del sistema, proporcionando interfaces para trabajar con éstos, aceptando tanto archivos binarios como bases de datos. El tercero trata con los invariantes, incluyendo la sintaxis de dichos invariantes y algoritmos relacionados con ellos. Final-

[1] `http://kimba.mat.ucm.es/paste`

mente, el cuarto módulo implementa una funcionalidad inspirada en técnicas de *mutation testing* [Off94, BM99, ABL05, HM09]. En este módulo, a partir de una especificación y un conjunto de invariantes, generamos automáticamente un conjunto de *mutantes* (sistemas que presentan alguna alteración respecto a la especificación original), utilizando diferentes operadores de mutación. De estos mutantes extraemos trazas que archivamos en logs, los cuales son analizados respecto a los invariantes. Podemos decir que, un conjunto de invariantes será mejor que otro si encuentra más errores entre estos mutantes. Como veremos más adelante, en [AMM09] se discuten las ventajas de utilizar estas técnicas en nuestro marco de trabajo.

En el segundo artículo dentro de esta sección, [AMN09a], se demuestra formalmente la corrección de la metodología introducida en [AMN08]. Para ello definimos en primer lugar una noción de corrección de una implementación respecto a una especificación. Dada esta noción de corrección, proponemos y demostramos el siguiente resultado: Dados una especificación formal del sistema, una implementación, y un conjunto de invariantes que son correctos con respecto a la especificación, si encontramos un error en alguna traza de la implementación con el conjunto de invariantes, entonces podemos decir que la implementación no es conforme a la especificación.

Además de la demostración de corrección de la metodología, en [AMN09a] se presentan dos algoritmos. El primero de ellos permite comprobar la corrección de los invariantes con respecto a la especificación. La idea intuitiva es que se recorre la especificación desde todos los estados comprobando si el invariante se cumple o no. El segundo de los algoritmos que presentamos es el de comprobación de errores en un log utilizando un conjunto de invariantes. En lo referente a la inclusión de restricciones temporales en los sistemas a testear, tenemos que señalar que aunque el uso de tiempos fijos es muy útil en la especificación de determinados sistemas, por ejemplo en mecanismos hardware, donde podemos suponer fácilmente que el tiempo se mide mediante ciclos de reloj, este modelo de representación temporal es poco expresivo para sistemas más complejos, más fieles a la realidad. Por ello, en [AMN09c] presentamos un estudio en el que la especificación del tiempo se realiza usando variables aleatorias. En sus invariantes también se expresa el tiempo utilizando variables aleatorias. A partir de un contraste de hipótesis se comparan los valores temporales que aparecen en los logs del sistema con los que deberían obtenerse a partir de las variables aleatorias que aparecen dentro de los invariantes. Nuestra herramienta PasTe permite definir en la actualidad variables aleatorias de diferente naturaleza: Dirac, uniformes, discretas y exponenciales.

El uso de métodos de mutation testing, nos ha permitido realizar un estudio pormenorizado de qué tipo de invariantes son mejores independientemente de cuál sea la especifica-

ción [AMM09]. La clasificación de los diferentes tipos de invariantes y sus características se ha realizado utilizando datos empíricos: Tomamos especificaciones generadas al azar y las mutamos con todos los operadores de mutación de todas las formas posibles, se extraen trazas de diferentes longitudes y se comprueba cuántos mutantes *mata* el conjunto de invariantes considerado. Un invariante mata un mutante si en las trazas generadas a partir del mutante detecta algún error. Los datos han sido analizados en base a diferentes criterios (operadores de mutación, longitud de la traza...) permitiendo establecer una preferencia a la hora de usar invariantes para detectar fallos, en base a su eficiencia. Para ello se propone una heurística para obtener de una forma rápida, mediante un método devorador, un conjunto *bueno* de invariantes.

La tercera parte de esta tesis se basa en aplicar la utilización de modelos probabilísticos para la representación de usuarios típicos del sistema y su posterior extracción de tests, presentados en la primera parte, con el marco teórico de testing pasivo de sistemas con información temporal presentado en la segunda parte de la misma. Los trabajos incluidos son: *Using a mining frequency patterns model to automate passive testing of real-time systems* [AMN09e], y *Supporting the extraction of timed properties for passive testing by using probabilistic user models* [AMN09d].

Un campo de estudio dentro de nuestra metodología de testing pasivo es el de la obtención de conjuntos de invariantes. Las técnicas que hemos propuesto en el segundo bloque consideran que un experto del sistema, o un testeador, proporciona el conjunto de invariantes a utilizar. Por ello, antes de analizar los logs respecto a dichos invariantes debemos asegurar que sean correctos con respecto al modelo formal de la especificación. El problema radica en que el número de invariantes que a un experto se le puede ocurrir con respecto a un determinado sistema puede ser pequeño, comparado con el número de invariantes que en el sistema se podría estar comprobando en cada momento. La capacidad tecnológica actual nos permite, sin provocar una sobrecarga elevada del sistema, comprobar un gran número de invariantes. Por ello, esta tercera sección de la tesis se centra en cómo obtener un conjunto de invariantes, de forma finita, rápida y automática, para utilizarlos en la tarea de testing pasivo. La línea común de los dos artículos que integran esta parte de la tesis consiste en buscar diferentes formas de proponer conjuntos de invariantes temporales.

En [AMN09e] se plantea que la principal desventaja de utilizar un algoritmo de extracción de invariantes a partir de una especificación, es que éste nos devuelve, en general, un conjunto infinito de invariantes, sin que dispongamos de ninguna medida para decidir cuáles son los mejores o cuál es el conjunto finito óptimo. Por ello, tanto en [AMN09e] como en [AMN09d],

se propone la inclusión de información extra dentro del marco de extracción de invariantes que ayude en esta tarea. De forma similar a como hicimos en los trabajos presentados en el primer bloque de la tesis, consideramos que partimos de un sistema que queremos testear, y que somos capaces de guardar las interacciones que hacen los distintos usuarios con esa máquina. Después de tener un gran número de acciones de los usuarios, somos capaces de generar de manera automática un modelo probabilístico para representar el conjunto de usuarios del sistema. La idea es generar el conjunto finito de caminos dentro del modelo probabilístico de usuario que represente el $\alpha\,\%$ de los comportamientos del sistema. La intención es testear, mediante testing pasivo, las acciones (o secuencias de acciones) más frecuentes.

Teniendo en cuenta el conjunto finito de interacciones de los usuarios con el sistema, en [AMN09d], se propone una adaptación del algoritmo presentado en [AMN09e]. El algoritmo recibe el modelo de la especificación, un modelo de usuario del sistema y un grado de representación α, y devuelve un conjunto finito de invariantes, correctos con respecto a la especificación, con $\alpha\,\%$ de representación.

En esta sección de la tesis también se recoge la extensión de PasTe para poder llevar a la práctica los nuevos métodos. Para ello hacemos uso de algoritmos de data mining, que son capaces de generar los modelos de usuarios de una manera rápida y eficaz. Por otro lado, se han incluido en la herramienta tanto el algoritmo de extracción directa de invariantes desde la especificación, como el algoritmo de extracción de invariantes con respecto al modelo de usuario.

La última parte de la tesis comprende dos casos de estudio realizados en el ámbito de los protocolos. Los trabajos incluidos en esta sección son: *Analysis of the OLSR Protocol by using formal passive testing* [AMC$^+$09], y *Formal passive testing of timed systems: A case study with the Stream Control Transmission Protocol* [AMN09b].

El estudio de casos industriales es una de las peticiones más demandadas para mostrar la aplicabilidad de las metodologías de testing desarrolladas. En nuestro caso, hemos tenido que adaptar los marcos teóricos propuestos inicialmente, sustituyendo las máquinas de estados finitos con acciones de entrada y salida, por autómatas temporales [AD94], y adaptando nuestros invariantes temporales a dicho formalismo.

En [AMC$^+$09] estudiamos el protocolo OLSR. Se trata de un protocolo muy utilizado dentro del grupo de investigación de Telecom & Management SudParis, dirigido por la profesora Ana Cavalli. Durante mi estancia en París en el seno de su grupo de investigación, pude aprender sus principales propiedades y estudiar diversas aplicaciones reales. En concreto el protocolo citado sirve para tratar redes *adhoc*. Una red adhoc no tiene una estructura está-

tica definida, sino que se va construyendo a medida que se van incorporando nuevos nodos. Cuando un nodo nuevo quiere empezar la comunicación dentro de un grupo ya formado, tiene que mandar unos mensajes predefinidos a alguno de los nodos ya existentes. A continuación se incluye el nuevo nodo en la lista de vecinos del antiguo nodo. Hay que tener en cuenta que dependiendo de cómo se vean unos nodos a otros, del modelo de enrutado que se construya dinámicamente, y de propiedades de rendimiento del entorno, los tipos de enlace que existen entre los nodos van cambiando.

Para poder adaptar el trabajo previo sobre testing pasivo de sistemas con información temporal, el primer paso fue la definición del tipo de mensaje que se envía por el medio, así como la representación de todos los campos que nos interesaban de éste, como eran su IP de origen, su IP destino, un conjunto de nodos vecinos, una serie de enlaces que podría tener asociado, una serie de nodos por cada enlace, etc. Asimismo, después de la definición de los mensajes, fue necesario definir una noción de *similitud* entre mensajes. Es decir, una forma de expresar la "equivalencia" de los mismos.

La única *especificación* disponible de este protocolo era textual, careciendo por tanto de un modelo formal. No obstante pudimos extraer un gran número de requisitos, tanto a nivel de seguridad como de integridad, que se debían cumplir en cada momento. Se instaló el protocolo OLSR en cuatro ordenadores portátiles diferentes. Al comienzo de los experimentos sólo dos de ellos estaban en comunicación directa. Según avanzaba el experimento, los otros dos ordenadores iban acercándose y alejándose de la red. Se utilizó un único interfaz de red para grabar todos los eventos que veía (recordemos que las comunicaciones son vía WIFI y que, por tanto, todos los mensajes son visibles). A partir del log del sistema, obtenido mediante el interfaz de red que estaba escuchando, y la lista de requisitos, decidimos aplicar nuestra metodología de testing pasivo para comprobar si conseguíamos testear correctamente todos los requisitos anteriores. Para ello, lo primero que hicimos fue la traducción de los requisitos a invariantes temporales. Algunas peculiaridades de los requisitos nos llevaron a proponer un nuevo tipo de invariantes que no habíamos considerado en trabajos anteriores. Con este nuevo tipo de invariantes conseguimos que todos los requisitos se pudieran formular de manera adecuada.

En [AMC+09] también realizamos un estudio para comprobar, de forma experimental, cuál de las dos nociones de invariante nos proporcionaba mejores resultados. Dado que analizábamos una implementación correcta del protocolo, no detectamos errores de comportamiento, pero sí fuimos capaces de detectar algunos errores de rendimiento, que eran difícilmente identificables.

En cuanto al segundo caso de estudio [AMN09b], éste se realizó sobre el protocolo SCTP. En este caso sí disponíamos de una representación formal de la especificación mediante un autómata temporal. El problema que nos encontramos con esta especificación fue que era demasiado permisiva en cuanto a valores de rendimiento. Es decir, se centraba en que la transmisión funcionara, sin valorar mucho los tiempos de respuesta de los nodos. Nosotros propusimos la inclusión de invariantes expresando propiedades temporales más restrictivas que la especificación, pero cuyo comportamiento funcional seguía siendo correcto. Al igual que en el caso anterior, tuvimos que modificar nuestro formalismo para representar de manera adecuada los mensajes que aparecían en la traza. En [AMN09b] la aplicación de técnicas de testing pasivo se realizó sobre trazas reales grabadas desde un único ordenador. Estas trazas tenían diferente longitud y su comprobación fue ejecutada en modo offline.

Cabe destacar que tanto en [AMC+09] como en [AMN09b] la monitorización de las trazas se llevó a cabo mediante el uso de la herramienta de monitorización Wireshark[2]. Esta herramienta facilita métodos de filtrado de mensajes por tipo de protocolo, muy útiles a la hora de abstraer el protocolo bajo estudio del resto de los mensajes que se han observado.

Como conclusión, podemos afirmar que la utilización de nuestra metodología a la hora de analizar protocolos de telecomunicaciones reales es en efecto viable.

Dentro de los objetivos de esta parte de la tesis, además de considerar casos de estudio reales, nos centramos en la aplicabilidad de PasTe en ámbitos no académicos. Durante la realización del proyecto *Verificación de requisitos de sistemas mediante máquinas de estados finitas utilizando un software de monitorización* con la empresa Peopleware S.L. (Contrato de Investigación, Artículo 83), que se desarrolló desde Diciembre de 2008 hasta Octubre de 2009, hemos extendido PasTe, permitiendo su integración dentro de la herramienta de monitorización OSMIUS. Esta herramienta se encarga de monitorizar, mediante un paradigma de agentes, cualquier instancia que se quiera inspeccionar dentro de una red computacional, ya sea una base de datos, rendimiento de CPU, utilización de la red, utilización de disco, memoria RAM, etc. El proyecto se centró en la inclusión de un módulo de verificación en tiempo de ejecución de las tareas monitorizadas. Durante el proyecto, se trabajó en un marco donde no disponíamos de un modelo formal de la especificación del sistema, pero sí de un conjunto de reglas que se debían comprobar. La sintaxis original de los invariantes en PasTe se modificó para representar estos conjuntos de reglas, permitiendo que los usuarios del sistema pudieran testear fácilmente las propiedades en que estuvieran interesados. Para ello, se proporciona a los usuarios un interfaz gráfico para representar los invariantes de una

[2] `http://www.wireshark.org/`

forma sencilla. Actualmente estamos estudiando la posibilidad de utilizar técnicas de lenguaje natural para, con ayuda de una ontología, introducir dichos requisitos dentro del sistema, en forma de frases enunciativas afirmativas.

Esta tesis ha permitido conformar un todo, a mi juicio bastante sólido, que estimo ha contribuido a enriquecer notablemente la colaboración real entre el mundo de las ideas, a un nivel teórico, con la empresa privada que facilita las aplicaciones prácticas; contando en el primer caso no sólo con la colaboración de mis directores y otras personas del grupo, sino con las de otros institutos de investigación.

CAPÍTULO 2

Estado del Arte: Métodos Formales y Testing

Lo que sabemos es una gota de agua; lo que ignoramos es el océano.

Isaac Newton

Con el avance de la tecnología informática, las técnicas de apoyo a la producción de software han aumentado notablemente su relevancia en este área. Entre ellas, los *métodos formales* y el *testing* han adquirido especial significación.

El testing de los sistemas que se desarrollan en la actualidad supone un alto coste en el proceso de producción, llegando a destinarse más del 50 % del presupuesto total del desarrollo de un sistema a la detección de errores. Uno de los aspectos más costosos asociado con las tareas de testing es el trabajo que debe desarrollarse manualmente, por lo que la mejora de dicho proceso se ha orientado a la automatización del mismo. Han habido ya numerosas propuestas, basadas en la combinación de los métodos formales y técnicas de testing, enfocadas a dicha finalidad. Tradicionalmente, los métodos formales y el testing no estaban muy vinculados, siendo la interacción entre ellos muy escasa. Sin embargo, durante los últimos años han llegado a ser complementarios, existiendo muchas metodologías de testing en las que la generación de tests se basa en la especificación formal de los sistemas.

Por otra parte, existen muchas técnicas de testing que tratan de aumentar la calidad del producto final mediante el incremento de su nivel de *efectividad* y *eficiencia*. La efectividad se optimiza mediante el uso de conjuntos de tests que proporcionen una alta probabilidad de detectar fallos, mientras que la eficiencia se incrementa reduciendo el número de tests que deben ser ejecutados para alcanzar dicho objetivo.

Este capítulo presenta una revisión de los lenguajes utilizados para especificar formalmente sistemas para su testeo, así como las principales contribuciones en el área de testing formal para dichos lenguajes.

2.1. Lenguajes formales de especificación

Los lenguajes de especificación *"informal"* aplican una combinación de gramáticas semiformales, texto libre y representaciones gráficas para describir los requerimientos de los sistemas, por lo que la precisión de las especificaciones se ve afectada por diferentes factores, entre ellos la experiencia del especificador. Sin embargo, los requerimientos de un sistema pueden ser representados mediante el uso de *lenguajes de especificación formales*, que permiten evitar las ambigüedades usualmente asociadas con las especificaciones informales.

Los lenguajes formales tienen como propósito facilitar una descripción rigurosa de los sistemas que se desea desarrollar, así como el análisis de su comportamiento. En los últimos treinta años los métodos formales han alcanzado un nivel de madurez que permite que sean aplicados con alta frecuencia a lo largo de todo el ciclo de vida del desarrollo de un sistema, en especial, cuando los sistemas requieran un alto nivel de seguridad.

Un lenguaje formal utiliza una *sintaxis* que permite describir de forma precisa la especificación de los sistemas, sea textual o gráficamente. También facilita una *semántica* que proporciona un significado preciso de cada descripción de los sistemas para los que el lenguaje se usa. La especificación de un sistema suele contemplar diferentes aspectos del mismo, incluyendo su comportamiento funcional, su estructura o arquitectura, e incluso aspectos no funcionales, como propiedades temporales o de rendimiento, por lo que la elección de un lenguaje u otro vendrá determinado por la adecuación del mismo a las características de cada sistema.

A continuación se revisan los principales lenguajes de especificación formales usados en marcos formales para realizar testing y su aplicabilidad a los diferentes tipos de sistemas.

2.1.1. Lenguajes basados en modelos

Hay diferentes modos de describir la especificación de un sistema, siendo uno de ellos la construcción de un modelo del comportamiento previsto. Lenguajes como Z [Spi88, Spi92], VDM [Jon91] y B [Abr96] permiten describir los estados del sistema junto con las operaciones que provocan el cambio de estado. En estos lenguajes los estados de un sistema suelen describirse mediante conjuntos, secuencias, relaciones y funciones, mientras que las operaciones lo hacen mediante predicados en términos de condiciones pre-post.

2.1.2. Lenguajes basados en estados finitos

Los lenguajes basados en modelos permiten describir sistemas generales, en particular, sistemas que potencialmente pueden presentar infinitos estados. Este carácter general tiene como inconveniente que el razonamiento sobre los mismos sea menos susceptible a la automatización. Los lenguajes de especificación basados en estados finitos, como su nombre sugiere, permiten la definición de un conjunto finito de estados, que suelen ser representados gráficamente mediante transiciones, que indican los cambios entre dichos estados. Entre estos lenguajes cabe destacar las *máquinas de estados finitos* (FSM) [LY96], SDL [CCI88, ITU92], *Statecharts* [Har87, HG97] y las *X-machines* [HI98]. La mayor parte del trabajo desarrollado con FSMs en el área de testing ha sido motivado por el testing de protocolos de comunicación, ya que las FSMs son muy apropiadas para la especificación de la estructura de control de los mismos. Con posterioridad, el testing basado en FSMs ha sido aplicado en el área de *testing basado en modelos*, en el que las especificaciones se utilizan para dirigir el proceso de testing.

En muchos casos, estos lenguajes de especificación permiten la representación de datos

internos adicionales; las operaciones representadas por las transiciones pueden acceder a estos datos y modificarlos, pudiéndose establecer también condiciones sobre los mismos. Las maquinas de estados finitos que presentan estas características son conocidas como *máquinas de estados finitos extendidas* (EFSM).

Otro formalismo de especificación que permite la integración de la estructura de control y los datos de un sistema son las X-machines. Estas máquinas disponen de una *memoria interna* y un conjunto de *funciones de proceso* que etiquetan las transiciones entre estados. Dichas funciones están definidas sobre el conjunto de entradas posibles y los valores de la memoria, produciendo en cada caso la salida esperada. Este formalismo fue introducido en [Eil74], y posteriormente propuesto en [Hol88] como lenguaje de especificación. Se han estudiado y desarrollado diferentes tipos de X-machines basados en restricciones sobre el conjunto de funciones y datos, siendo las stream X-machines [Lay92] las que han recibido una mayor atención.

2.1.3. Lenguajes basados en álgebras de procesos

Las *álgebras de procesos*, véase [BPS01] para tener una visión panorámica del campo, permiten describir sistemas con procesos concurrentes, donde varios procesos interactúan comunicándose entre ellos. Entre estos lenguajes se incluyen CSP [Hoa85], CCS [Mil80, Mil89], ACP [BW90], π-*calculus* [MPW92, Mil99, AG99] y LOTOS [LOT88].

2.1.4. Lenguajes algebraicos

Aunque las álgebras de proceso son adecuadas para la manipulación algebraica, hay lenguajes que describen los sistemas en términos de sus propiedades algebraicas, mediante axiomas y reglas que caracterizan completamente las propiedades deseadas. Ejemplos de estos lenguajes son OBJ [GT79] y CASL [BM04, Mos04].

En términos matemáticos, un álgebra consiste en un conjunto de símbolos que denotan valores de algún tipo y un conjunto de operaciones sobre los mismos. Para describir las reglas que gobiernan el comportamiento de las operaciones es necesario especificar la sintaxis y la semántica de las operaciones. En este tipo de lenguajes, la primera usa una signatura para cada operación, indicando su dominio y rango. En cuanto a la semántica, ésta viene dada mediante ecuaciones que de modo implícito describen las propiedades requeridas.

2.2. Técnicas de testing

El desarrollo de software ha pasado de ser un proceso constituido por pocas tareas en las que intervienen pocas personas, a ser un proceso muy complejo en el que participan grandes equipos. Al pasar a involucrar a todos los participantes en cada una de las etapas de desarrollo, se ha convertido en una necesidad la buena planificación de esta tarea en el ciclo de vida de la producción de un sistema.

Cuando se utiliza un lenguaje de especificación formal, se puede aplicar un análisis formal en las fases de especificación, diseño y codificación. Ello puede suponer tan sólo la comprobación de ciertas propiedades, o el establecimiento formal de la conformidad de un sistema con respecto a su especificación. Estas demostraciones se pueden realizar, en algunos casos, de un modo automático, reduciéndose así la probabilidad de incurrir en errores humanos.

El proceso de testing implica la ejecución de la implementación que debe ser chequeada: se aplica una secuencia de entradas al sistema y se observan las salidas recibidas. Esta relación entre entradas y salidas permite establecer la adecuación del sistema testeado a la especificación de la que partimos. Las secuencias de entradas aplicadas a la implementación se llaman *tests*. Entre las diferentes técnicas de testing se puede distinguir entre el testing de *caja negra* y el testing de *caja blanca*. En el *testing de caja negra* un sistema se chequea sin que se conozca su estructura interna. Los tests se generan a partir de la especificación y sólo se dispone de información acerca de las salidas esperadas para las entradas aplicadas. Ya que no se va a contar con ninguna información sobre cómo ha sido desarrollado el sistema, los tests se pueden generar a partir del momento en que la especificación está disponible. Otra categoría es el *testing de caja blanca*, que utiliza información de la estructura interna de la implementación para la generación de tests. En este tipo de testing, se pueden generar tests con el fin de chequear características específicas del sistema.

Además de poder argumentar si una técnica de testing puede establecer la conformidad de una implementación respecto a una especificación mediante un determinado conjunto de tests, hay otras propiedades de dichas implementaciones que se pueden establecer. Esta idea se formaliza en [GG75] mediante las nociones de *corrección y completitud*.

Un conjunto de tests es correcto si es capaz de detectar la incorrección de cualquier sistema erróneo. Por otra parte, un conjunto de tests es completo si el hecho de que un sistema supere la aplicación de todos los tests del mismo permite deducir la corrección del sistema. Por tanto, si el conjunto de tests derivado mediante una técnica específica de testing es correcto y completo, su aplicación determinará la corrección o incorrección de una implementación. En general, para que una técnica de testing tenga estas dos propiedades

se requiere la generación de un conjunto infinito de tests, lo que no permite su aplicación práctica y hace necesario aplicar un criterio de selección que permita reducirnos a la aplicación un conjunto finito de tests. En la mayoría de los casos, los criterios de selección se basan en *hipótesis* como "si el sistema es correcto para un subconjunto de los valores de entrada permitidos, entonces es correcto para todos" o "si un sistema es correcto para un valor de todos los que pueden ser aplicados en una determinada secuencia, entonces es correcto para todos los valores posibles". Aunque hay muchas hipótesis posibles para la selección de tests [Gau95] se pueden identificar dos tipos principales: *hipótesis de uniformidad e hipótesis de regularidad*. El primer tipo hace referencia a la uniformidad del comportamiento de un sistema sobre un rango de datos. El segundo tipo se relaciona con la regularidad del comportamiento del sistema cuando el tamaño de los datos aumenta. Por ejemplo, se podría asumir que si un sistema funciona correctamente para un buffer de tamaño 0, 1, y 2, entonces lo hará cualquiera que sea el tamaño del mismo. La idea de hipótesis está muy relacionada con la noción de *modelo de fallos* [IT97]: un conjunto de modelos entre los cuales se encuentra uno (desconocido) funcionalmente equivalente a la implementación testeada.

Una vez establecido el conjunto de hipótesis o el modelo de fallos más apropiado, la técnica de testing debe generar un conjunto de tests que permita establecer la corrección de los sistemas respecto a las especificaciones, asumiendo que se cumplen dichas hipótesis.

A continuación se revisan diversas metodologías de testing propuestas para los diferentes formalismos previamente presentados.

2.2.1. Testing para especificaciones formales basadas en modelos

Hay muchos trabajos de testing orientados a especificaciones Z, B, y VDM. Las técnicas de generación de tests para ellos están basadas en hipótesis de uniformidad. El dominio de los valores de entrada es particionado en subdominios en los que el comportamiento del sistema se asume uniforme. Si la hipótesis de uniformidad se cumple, basta con aplicar un dato de cada clase para testear el sistema. Muchas de las técnicas de derivación de tests propuestas en este ámbito han sido automatizadas; en ellas son aplicadas diferentes heurísticas para realizar la partición en clases de equivalencia y la selección de los datos que se utilizarán durante el proceso de testeo.

[Hal89] propone un método de testing para especificaciones Z basado en una clasificación en dominios de tests. La idea se basa en considerar diferentes combinaciones de subdominios obtenidos mediante la partición de los conjuntos de entradas, salidas y estados, y las condiciones contenidas en los predicados de la especificación. [DF93] demuestra que, mediante

la reescritura de la precondición y la postcondición de una especificación en forma normal disyuntiva, una gran parte del proceso de análisis de partición puede ser automatizado. Así mismo, los autores describen una técnica para generar a partir de la especificación, un autómata finito que puede ser usado para dirigir el proceso de generación de tests. La combinación de partición de categorías y forma normal disyuntiva fue aplicada para las generación de tests en [SCS97].

Una propuesta completamente diferente es la *mutación de especificaciones* [CS94]. La idea está inspirada en la técnica de testing mediante *mutación* de programas [How82, Off94, BM99, ABL05, HM09]. La aplicación de *operadores de mutación* a la especificación genera mutantes. Para cada mutante obtenido se genera un test que distinga el comportamiento de éste y la especificación. La hipótesis en este caso es que si un conjunto de tests obtenido mediante este método permite distinguir entre un mutante y la especificación, también distinguirá entre la especificación y una implementación incorrecta.

2.2.2. Testing basado en máquinas de estados finitos

Muchos sistemas tienen una estructura de estados finitos y por ello las FSMs son un formalismo adecuado para su representación. Por ello, el testing de FSMs ha recibido mucha atención. Moore planteó el marco conceptual de testing basado en FSMs en su trabajo [Moo56] referido a un concepto muy utilizado en la física, los *experimentos gedanken*. Los principios fundamentales del chequeo de transiciones fueron enunciados en [Hen64], trabajo motivado por el testing de circuitos secuenciales, que más tarde fue aplicado al testing de protocolos de comunicación [LY96].

Las FSMs definen un lenguaje relativamente pobre en cuanto a expresividad y técnicas de abstracción. Sin embargo, la falta de expresividad tiene grandes ventajas cuando se analizan las FSMs. Muchos problemas que no son decidibles en marcos más generales lo son para FSMs, y frecuentemente además con complejidad polinomial, lo que facilita la automatización de la generación de tests. Como se dijo anteriormente, muchas propuestas para testing basado en FSMs consideran o *hipótesis* o un *modelo de fallos*. En ambos casos, el conjunto de tests garantiza la determinación de la corrección de los sistemas, siempre que se cumplan las hipótesis consideradas. En el testing basado en una especificación representada mediante una FSM, es normal asumir que la implementación es equivalente a una FSM desconocida, y que el conjunto de tests generado permite chequear si la implementación es *conforme* a la especificación. Si la especificación es determinista, la noción de conformidad coincide con la equivalencia de especificación e implementación. Si la especificación es no determinista,

hay nociones alternativas de conformidad, como considerar que el comportamiento de la implementación es un *subconjunto* del comportamiento de la especificación.

FSMs completamente especificadas y deterministas

En primer lugar trataremos el caso de las FSMs completamente especificadas, mínimas y deterministas, esto es, máquinas donde para cada entrada disponible y cada estado existe una única transición etiquetada con dicha entrada, no tienen estados equivalentes y toda entrada tiene una transición asociada en cada estado. En este caso la relación de conformidad entre la especificación y la implementación es la *equivalencia*, es decir, para las mismas secuencias de entradas se producen las mismas secuencia de salidas. El método de recorrido de transiciones [NT81], abreviado como método TT, produce un conjunto de tests que ejecuta todas las transiciones de la especificación. Este método sólo está orientado a la detección de fallos de salida. La variante del método presentada en [SMIM90] tiene menos capacidad de detección de fallos: cubre todos los estados, pero no necesariamente todas las transiciones.

El método D [Hen64, Gon70, Koh78] asume como hipótesis que la implementación no tiene más estados que la especificación. El método chequea todas las transiciones con el fin de localizar tanto fallos de salida como de transición de estados. Este método ha sido optimizado para reducir el número de tests necesarios [UWZ97, HU02].

Otra propuesta está basada en secuencias únicas de entrada/salida (UIO) [SD88]. En este caso las hipótesis son, por una parte, que la implementación no tiene más estados que la especificación y, por otra, que existe un *reset* que lleva la implementación al estado inicial. Las secuencias UIO se usan para verificar que tras una serie de interacciones, la máquina se encuentra en el estado esperado.

Cuando no se cumple la hipótesis de que el número de estados de la implementación no supera al de la especificación, se puede aplicar el método W [Vas73, Cho78]. En este caso se considera que existe una cota superior en el número de estados de la implementación y un reset correctamente implementado. Bajo estas condiciones, el método proporciona cobertura total de fallos. Este método genera el conjunto de tests basándose en un *conjunto de caracterización* y un *conjunto de cobertura de estados*.

FSMs parciales

La mayoría de las especificaciones reales son *parciales*, por lo que no cubren todas las posibles combinaciones de estado/entrada. Debido a las diferentes interpretaciones de las *transiciones no definidas*, han sido consideradas diversas semánticas [BP94]. Puede consi-

derarse que estas transiciones están implícitamente definidas, es decir, se sustituyen por transiciones con el mismo estado de origen y destino con salidas nulas, o bien por transiciones que llevan a un estado de error y producen una salida de error. De este modo se obtiene una FSM completamente especificada y se pueden aplicar las estrategias presentadas anteriormente. Otra posible interpretación considera que son transiciones prohibidas, por lo que no deberían ser ejecutadas por la implementación. En consecuencia, el conjunto de tests no debe considerar estas transiciones.

FSMs no deterministas

Hay diferentes motivos que pueden ocasionar la presencia de indeterminismo en una especificación. El no determinismo puede representar un requerimiento para el sistema especificado, en cuyo caso la conformidad de la implementación corresponderá a la equivalencia de la misma respecto a la especificación. Sin embargo, la presencia de no determinismo puede representar simplemente un abanico de opciones, en cuyo caso será suficiente que los comportamientos de la implementación sean un subconjunto de los admitidos por la especificación.

El no determinismo presenta una cuestión a tener en cuenta: La observabilidad de cada posible repuesta asociada con una secuencia de entradas en particular. Con el fin de abordar este problema es frecuente asumir la hipótesis de que existe una cota n, tal que si una misma secuencia de entradas ha sido aplicada n veces en un estado, entonces queda garantizado que todas las posibles salidas asociadas con dicho estado y secuencia de entradas han sido observadas. Esta hipótesis se conoce como *fairness hypothesis*.

FSMs extendidas

Una máquina de estados finitos extendida es una FSM con parámetros de entrada y salida, variables auxiliares y operaciones y predicados definidos sobre las variables y los parámetros de entrada. Si se aplica una hipótesis de uniformidad a los datos es posible generar un conjunto de tests abstrayendo los citados parámetros de la EFSM. En consecuencia, el proceso de testing estará orientado a la estructura de control de la implementación [DU04]. Si no se asume la hipótesis de uniformidad, entonces el proceso de testing afectará tanto a la estructura de control como a la estructura de datos. Estos dos componentes se testean independientemente, aplicando métodos basados en FSMs para la parte de control y métodos de testing para el flujo de datos a la parte correspondiente a los mismos [USW00]. Bajo hipótesis técnicas precisas, una EFSM se puede transformar en una FSM equivalente. Aunque estas transformaciones conllevan frecuentemente un problema de explosión de estados, en

algunos casos se concreta con métodos efectivos para realizar la conversión [PBG04].

Una estrategia alternativa está basada en el uso de conjuntos de *propósitos de test*. Un propósito de test es una descripción de un requerimiento para un test, como por ejemplo, la ejecución de una transición o una secuencia de transiciones en particular, y puede representarse mediante un autómata de estados finitos. Para cada propósito de test se genera un test que lo satisface. El test puede construirse automáticamente, aplicando análisis de alcanzabilidad al producto del propósito de test y la especificación. Aunque el análisis de alcanzabilidad sufre del problema de explosión de estados, hay herramientas que aplican una estrategia *on-the-fly*, como TGV, que son efectivas en la práctica [KJG99]. Usualmente los propósitos de test son proporcionados por el testeador, pero pueden obtenerse también automáticamente a partir de la EFSM que representa a la especificación, teniendo en cuenta un cierto criterio de test establecido, como podría ser la ejecución de todas sus transiciones. Otro tipo de propósito de test, más restringido, consiste en requerir una secuencia de interacciones que puede describirse mediante un *message sequence chart* (MSC). Existen herramientas como AUTOLINK que aplican un análisis de alcanzabilidad al producto de la especificación y un MSC, para producir un test apropiado [SEK+98].

Finalmente, se ha abordado también el problema de testing mediante la conversión en FSM equivalentes de especificaciones realizadas en el contexto de otros modelos formales, con el fin de aplicar los métodos de testing basados en dicho formalismo. Entre ellos se incluyen métodos de conversión de variantes de especificaciones Z [DF93, Hie97, DB99], Statecharts [HSS01] y SDL [BPBM97].

Stream X-Machines

El primer método de testing basado en Stream X-Machines se propuso en [IH97, HI98] para sistemas deterministas. Esta técnica de testing genera un conjunto de tests a partir de una especificación, asumiendo que las funciones de proceso asociadas a las transiciones están correctamente implementadas. Se requiere además que el conjunto de funciones de la especificación y la implementación coincidan, y que se satisfagan dos condiciones, habitualmente denominadas *design for test conditions*. Estas condiciones consisten en la distinguibilidad de las funciones de proceso mediante las salidas emitidas, y la seguridad de que mediante las entradas apropiadas, dichas funciones podrán ser chequeadas en la implementación. Este método ha sido generalizado en [Ipa04] para suprimir la hipótesis de la corrección de la implementación de las funciones de proceso. Así mismo, se relaja la restricción de que el conjunto de funciones deba coincidir en la especificación y la implementación.

También existen propuestas para máquinas no deterministas. En [IH00] se extiende el método para cubrir máquinas no deterministas y se presenta una técnica de testing en la que la noción de corrección es la equivalencia de máquinas. Otra metodología alternativa ha sido propuesta en [HH00, HH04], en ella se hace uso del método de testing conocido como *state counting*, y se considera una noción de conformidad basada en que los comportamientos de la implementación son un subconjunto de los de la especificación.

2.2.3. Testing para álgebras de procesos

El principal estudio de testing en este área fue desarrollado por de Nicola y Hennessy [dNH84, Hen85, Hen88]. Ellos introdujeron diferentes *preordenes y equivalencias* para relacionar procesos mediante su interacción con conjuntos de tests. Esencialmente se distinguen dos familias de relaciones: *may* y *must*. En las primeras se permite que el proceso pase el test con éxito en alguna ejecución; sin embargo en la segunda categoría ello debe ser así en todas las ejecuciones posibles. El marco original ha sido extendido para tratar con propiedades no funcionales como tiempo [HR95, LdF99], probabilidades [LS89, Chr90, NdF95, Núñ03], y una combinación de ambas [GLNP97]. Los *sistemas de transiciones etiquetadas* (LTS) se usan frecuentemente para describir la semántica de las álgebras de procesos. Las técnicas de testing para LTSs se basan en relaciones de conformidad y existen numerosos algoritmos de generación de conjuntos de tests basados en diferentes relaciones de conformidad. Entre ellos se pueden citar [Led91, Pha94, Tre96].

2.2.4. Testing para especificaciones algebraicas

El primer trabajo que empleó especificaciones algebraicas en una metodología de testing fue el desarrollado para el sistema DAISTS [GMH81]. Sin embargo, las propuestas más significativas se presentan en [GJ98, Gau01].

Es evidente que la aplicación de un conjunto de tests exhaustivo que no muestra fallos garantiza la corrección del sistema. No obstante, la naturaleza infinita de este conjunto de tests lo hace impracticable, por lo que se requiere limitar este conjunto mediante hipótesis de regularidad y uniformidad. Las hipótesis de regularidad, en el contexto de las especificaciones algebraicas, se basan en el número de constantes y constructores que aparecen en un término. La hipótesis permite establecer la corrección del sistema si éste funciona correctamente para tests de hasta un cierto tamaño n. Las hipótesis de uniformidad permiten de nuevo establecer si el sistema funciona correctamente para una selección de valores dentro de los diferentes subdominios establecidos mediante partición. El método utilizado más frecuentemente para

la selección de los valores es *Boundary Value Analysis* [WC80, CHR82, LPU02].

Hay dos posibilidades para generar tests a partir de especificaciones algebraicas: usando la sintaxis de las operaciones o los axiomas. El primer método se presento inicialmente en [Jal83] y posteriormente se ha aplicado en diferentes experimentos [JC88, Woo93, AW96]. El segundo método se introdujo en [GMH81] y ha sido utilizado en muchas propuestas posteriores [Cho86, DF94, CTCC98, CTC01].

CAPÍTULO 3

Estado del Arte: Testing Pasivo y Runtime Verification

No veo lógico rechazar datos porque parezcan increíbles.

Fred Hoyle

Las metodologías de testing se pueden clasificar en dos grandes clases: *testing activo* y *testing pasivo*. La existencia/ausencia de fallos en el testing pasivo se determina *observando* los *log* del sistema. El testeador es un mero observador del sistema que no puede interactuar con él, teniendo que decidir si el comportamiento observado es correcto o no. En otras palabras, los datos en tiempo de ejecución se utilizan en testing pasivo para determinar si el sistema que los produce cumple ciertos requisitos. Estos requisitos pueden referirse a relaciones de acción-reacción, esto es, de causa-efecto o relaciones causales entre distintos eventos.

Existen numerosas técnicas para estudiar la corrección de un sistema respecto a ciertos requisitos expresados con una lógica formal. En este paradigma englobamos las técnicas de *testing pasivo con invariantes*, principal objeto de esta tesis, y las utilizadas en *runtime verification*. La principal diferencia entre estas dos familias viene dada por la forma en que se definen las propiedades. En testing pasivo con invariantes se utilizan lógicas orientadas al testing, para las que es muy fácil expresar condiciones con entradas y salidas del sistema. Por otro lado, en runtime verification son muy utilizadas las lógicas temporales de carácter general. Estas técnicas son muy útiles en la fase beta de los programas, es decir en la fase inmediatamente anterior a la implantación definitiva del producto. En ella, todas las funcionalidades que el sistema provee están ya implementadas. Por tanto, los requisitos se deben cumplir en todos los logs generados por el sistema beta.

En este capítulo presentamos una revisión de los conceptos básicos del testing pasivo, y de las distintas clasificaciones de monitores que existen, así como una breve descripción de las principales técnicas del testing pasivo: técnicas de monitoring y técnicas de invariantes, junto con una descripción detallada de algunas técnicas relevantes de runtime verification.

3.1. Marco teórico

En la Figura 3.1 presentamos un esquema general de la aplicación de una metodología de testing pasivo sobre un sistema `P`. El sistema `P` se ejecuta en el contexto del entorno `E`. El medio `E` puede ser el sistema operativo, un middleware como EJB o JXTA, o algún otro marco de trabajo, como sistemas ubicuos o sistemas de red. El software recibe estímulos del usuario del sistema y en respuesta a ellos manda información sobre la forma en que ha variado su estado. Los eventos enviados por el usuario son las denominadas *entradas del sistema*, mientras que los eventos emitidos por el software son las *salidas del sistema*. Una parte muy importante dentro de este marco teórico es la obtención de eventos. Estos eventos se capturan monitorizando el sistema `P`. Los monitores se pueden encontrar dentro del sistema `P`, dentro

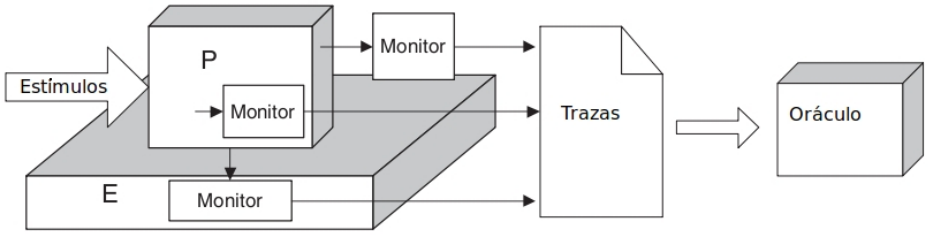

Figura 3.1: Escenario general de testing pasivo y runtime verification.

del entorno E, o externos a ambos. En la siguiente sección expondremos, con más detalle, las diferencias existentes entre algunas metodologías que sitúan monitores en uno u otro lugar. Los eventos se almacenan en los logs de ejecución, que serán los elementos de entrada para el *oráculo* (nomenclatura habitualmente usada en testing pasivo), que determinará la corrección de dicha ejecución.

Una de las condiciones a la hora de llevar a cabo las tareas de testing pasivo es consumir pocos recursos, más aun si éste se realiza de forma *online*, es decir, en tiempo de ejecución. Por ello, aquellas técnicas que realizan muchos cómputos a la hora de analizar los logs no son factibles, ya que conducen a un degradamiento del rendimiento del sistema. Cuando esto no sea posible, existen algunas alternativas para realizarlo "casi" online. Por ejemplo, consideremos un sistema que está ya implantado en una empresa, donde los usuarios finales ya están interactuando con él. Se podría diseñar un esquema de testing pasivo donde los datos se analizaran en primer lugar en los ordenadores personales de cada usuario. Acto seguido, se enviarían los resultados por red a un ordenador central, el cual se encargaría de comprobar propiedades más complejas casi en tiempo real. En este ejemplo, la degradación del servicio vendría dada por el número de mensajes que los terminales de los usuarios envían al servidor central, y por la capacidad de procesamiento de éste.

3.2. Monitores

La instalación de monitores es una actividad compleja, ya que el principal requisito de los mismos es que deben ser lo menos intrusivos que sea posible. Por otra parte, los monitores no deben estar localizados en un único punto del sistema, sino en diferentes lugares. De hecho,

```
class MyClass {
    static void foo(int x) {
        /*TRBegin
            TRAssert{ Always{x < 0} } =>
         {System.out.println("Violated the assertion x<0 in foo(int x), x=" + x);}
        TREnd*/
        }
    }
```

Figura 3.2: Ejemplo de aserción escrito en Temporal Rover.

una de las principales ramas de investigación de monitores es saber qué número de ellos se debería colocar y dónde. La tarea de instalación ya no se realiza manualmente, estando este procedimiento en la actualidad muy automatizado [DGR04].

Fijándonos de nuevo en la Figura 3.1, los monitores se pueden situar tanto a nivel interno de la aplicación `P`, como en el entorno `E` donde se ejecuta el sistema, como fuera del sistema a testear. En el caso de monitores instalados dentro del código, las técnicas más utilizadas se basan en la instrumentalización automática del código fuente o del código binario del programa. En estos casos se incluye código adicional en puntos estratégicos, por ejemplo en las llamadas a subrutinas, o antes de la ejecución de determinadas operaciones, para generar eventos [Ale07], o en código binario [SF07] o en algún lenguaje intermedio, como Java bytecode [dH05].

Relacionado con la ubicación de monitores dentro del sistema encontramos la posibilidad de incluir oráculos dentro del programa `P`. Esta situación surge frecuentemente durante el desarrollo de aplicaciones, al escribir *aserciones* en el código del programa [BY05]. Las aserciones se comprueban en tiempo de ejecución y si se detecta algún error, se genera un evento con la máxima información posible. La programación con aserciones, en contraposición con la programación defensiva, con múltiples instrucciones *if* intentando describir todos los casos, tiene una gran ventaja: en ella se permite la compilación del código fuente tanto con aserciones como sin ellas, de manera automática y trasparente (sólo es necesario cambiar las directivas de compilación). Normalmente, el programa se compila con las aserciones cuando se está desarrollando el sistema, mientras que, se utiliza una compilación sin ellas, en la versión final del producto.

En la actualidad existen muchos lenguajes que permiten la inclusión de este tipo de aserciones, y cada uno de ellos tiene un poder expresivo diferente. Por ejemplo, el *mecanismo de*

aserción de Java[1] permite la especificación de expresiones lógicas booleanas que pueden ser rápidamente comprobadas, mientras que *Temporal Rovers* [DS03] permite la especificación de aserciones utilizando lógicas temporales. En la Figura 3.2 se muestra un ejemplo de este tipo de aserciones introducidas en el código. Si se necesita especificar requisitos de tiempo real, podemos utilizar *Java Runtime Timing Constraint Monitor* [LMK07], que permite la especificación de aserciones en *lógicas de tiempo real* [FRS94]. Por otro lado, no sólo encontramos aserciones dentro del lenguaje Java, también se han descrito para otros lenguajes como Eifel [Mey92] (de hecho dicho lenguaje fue el primero que permitió la inclusión de aserciones), C [TJ98] y Ada [SM93].

En los casos en que el código fuente no sea accesible, es posible utilizar versiones modificadas del entorno `E` para monitorizar las ejecuciones. Ejemplos de ello son la utilización de una *Máquina Virtual de Java modificada* [Wol99] o el uso del entorno Dynascope [Sos92], donde se puede realizar una monitorización con grano muy fino de las operaciones, tanto a nivel de coste como de duración. Estas técnicas tienen la ventaja de que se puede monitorizar de forma automática cualquier aplicación que se ejecuta sobre ellas. Los principales inconvenientes son por un lado que no siempre podemos tener un entorno modificable, con las mismas prestaciones que el original; y por otro lado que la utilización de este tipo de entorno suponen un degradamiento en el rendimiento de otras aplicaciones que no necesitan ser monitorizadas. Es más, la modificación del entorno puede llevar a éste a ser incompatible con el entorno estándar. Finalmente, un monitor de entorno está programado de *manera general*, no es específico para cada programa. Ello implica que si se desea monitorizar determinadas partes de un sistema tenemos que programarlas directamente, ya que un monitor de carácter general no sería capaz de capturarlas.

La última clasificación de los monitores, respecto a su ubicación, es que no se encuentren ni dentro del programa ni en el entorno. Se han propuesto distintas técnicas, entre las que destacamos aquéllas en las que la monitorización se realiza sobre las redes que comunican distintos componentes con el sistema. Por ejemplo, se pueden diseñar monitores para capturar los eventos en la red [DJC94], o en la comunicación entre componentes [BS03]. Este último paradigma de monitorización tiene la ventaja de ser menos intrusivo, pero la desventaja de que solamente se monitorizan los elementos de intercambio de información. Otra desventaja, es que la implantación de este tipo de dispositivos es a menudo compleja [BS03].

[1] `http://testng.org/javadocs/org/testng/Assert.html`

3.3. Técnicas de testing pasivo

En esta sección empezaremos por hacer, en orden cronológico, un breve repaso de las técnicas más representativas de testing pasivo y monitoring. De hecho, dado que la distinción entre estos dos términos es mínima o incluso nula, en esta sección los usaremos de forma indistinta. Después nos centraremos en las técnicas de testing pasivo con invariantes, que es la aproximación concreta que seguimos en esta tesis.

3.3.1. Técnicas de monitoring y testing pasivo

Empezamos con los protocolos de comunicación y el testing pasivo. En [YWS+89] los autores presentan un sistema de monitorización y *gestor de fallos* sobre redes de telecomunicaciones. En este primer trabajo encontramos una discusión sobre las ventajas e inconvenientes de poder o no poder grabar toda la información del entorno, sobre el problema del espacio que esto generaría, y sobre cómo se debe hacer el filtrado de la información que se quiere grabar.

Además de contar con las observaciones del entorno, un gestor de fallos deberá contar con *información extra* para poder dar el veredicto más fiable a partir de lo que observa. La información extra que se requiere en [YWS+89] se clasifica en cinco tipos. El primero de ellos es el conocimiento de las *propiedades físicas* del medio donde se realiza la monitorización. Debemos ser capaces de incluir en el sistema, de forma veraz y sencilla, información relativa a las condiciones de los canales de comunicación entre los diferentes nodos. Imaginemos que estamos grabando el paso de mensajes por la red, y que a partir de un momento determinado no grabamos nada. Si nuestro gestor de fallos es capaz de recibir y procesar información del estilo "el cable de red está roto" entonces este tipo de fallos no se procesaría como un error de programación del sistema. Dentro del segundo tipo de información se engloban las reglas, algoritmos y metodologías *ligeras* utilizadas por los expertos a la hora de la observación y análisis de fallos. Una metodología ligera es aquélla que no utiliza muchos cómputos. Un ejemplo típico de tales reglas se tiene cuando un determinado nodo manda un mensaje a todos los nodos de la red, y ningún nodo de la red ha recibido dicho mensaje. En lugar de suponer que todas las conexiones de los nodos están mal, es más razonable asumir que el problema está en quien debía enviar el mensaje. El tercer tipo incluye aquellas técnicas capaces de representar las *experiencias pasadas*: por ejemplo, si varias implementaciones del protocolo han provocado un determinado fallo en redes parecidas a la nuestra, entonces es bastante probable que en nuestra red también se den este tipo de problemas. El cuarto grupo es parecido al anterior, pero no exactamente igual; consiste en la inclusión de *hechos*

predecibles. Si sabemos que en nuestra red se puede dar un determinado problema entonces, tendríamos que tenerlo muy en cuenta a la hora de razonar sobre las observaciones. El último tipo es el llamado *conocimiento oculto* del protocolo. Se engloban aquí las técnicas que sintetizan cómo trabaja el protocolo. Este conocimiento suele provenir de la especificación, de los documentos de diseño, y/o de los requisitos del sistema. Además del desarrollo teórico, en [YWS+89] se presenta la implementación de un gestor de fallos en una máquina SUN-3.

En [YWS+89] el número de observadores que hay en la red está restringido a uno, pero esto se solventa en [WS93], donde los autores presentan un mecanismo para la detección de fallos en entornos de comunicaciones donde se contemplan *múltiples observadores*. En el trabajo se propone una nueva metodología a la hora de identificar y detectar *fallos*. Se entiende por fallo toda aquella operación anormal que interrumpe o altera la comunicación, o degrada el rendimiento. Se utilizan las máquinas de estados finitos como modelo formal para especificar sistemas. Los observadores se crean dividiendo la especificación en varias máquinas de estados finitos, intentando que cada una de ellas represente un requisito concreto. Esta metodología no implica que un determinado fallo no se pueda detectar por varios observadores a la vez. En [WS93] se presenta también un algoritmo de testing pasivo para la detección de errores con múltiples observadores.

El siguiente trabajo a destacar es [LS94]. En el mismo, los autores consideran que, en tiempo de ejecución, se puede inferir *en qué estado se encuentra el sistema* con la utilización conjunta del log y la especificación. Se presenta un algoritmo que va comprobando el valor de las variables del log y calcula el conjunto de estados en los que la máquina puede encontrarse. Se presentan dos implementaciones eficientes de dicho algoritmo. La primera se centra en el rango posible de cada variable, obteniendo toda la información de las transiciones que aparecen en la especificación. En la segunda implementación se resuelve un conjunto de ecuaciones e inecuaciones lineales para determinar si las restricciones derivadas de un log de la implementación violan las restricciones fijadas por la especificación.

El trabajo propuesto en [LNS+97] utiliza testing pasivo para encontrar fallos en sistemas que implementan protocolos de comunicaciones. Se presentan algoritmos para testear sistemas especificados con máquinas de estados finitos, tanto deterministas como no deterministas. La idea en la que se basan estos algoritmos es la de intentar localizar en qué estado está la máquina mediante *homing sequences*, y a partir de ahí utilizar la especificación para comprobar la existencia/ausencia de errores. En el caso de máquinas no deterministas, se adapta el algoritmo anterior haciendo la suposición de que en lugar de estar en un estado en cada momento, "estamos" en un conjunto de estados. Para el caso de máquinas no ob-

servables, un caso particular de máquinas no deterministas, se presentan dos algoritmos de búsqueda en anchura, para calcular el posible estado de la implementación. En el primer algoritmo se introduce el número de acciones que no hemos observado, mientras que en el segundo se introduce una cota máxima respecto al número de acciones no observadas. Además del marco teórico, los autores aplican sus técnicas al protocolo SS7. Uno de los problemas de esta metodología es la imposibilidad de detectar los denominados *errores de cambio de estado*.

[LNS+97] ha sido un trabajo muy influyente y sus ideas claves han sido aplicadas en otros sistemas basados en máquinas de estados finitos [WZY01, ZYW03, UXZ07], en máquinas de estados finitos extendidas [TC99, LCH+02, ACC+04, LCH+06, UX07, BDS+07] y en máquinas comunicantes de estados finitos [Mil98, MA00, MA01].

Otra forma de realizar testing pasivo es mediante la utilización de métodos de *comprobación de firma* [NVN98]. Los autores proponen analizar la cantidad de información del log que se va a testear. El algoritmo clásico de firma asocia a cada traza de la especificación un valor. Este valor se guarda dentro de una tabla de firmas de forma estática. Durante la fase del testing, se computa el valor de la traza obtenida, y se comprueba que dicho valor existe dentro de la tabla estática.

Al precisar la aplicación de [NVN98], encontramos tres dificultades significativas. La primera es el tamaño de la tabla de firmas estática, pues en ocasiones el número de trazas de la especificación puede ser infinito. El segundo es el problema de acceso a los datos, dado que según aumenta el número de datos que se encuentran dentro de la tabla, más costoso es encontrar uno determinado. El tercer problema radica en la semántica que se le asocia a cada traza. En este sentido, los autores se plantean que si la firma asociada a cada traza tuviera significado por sí misma, esto ayudaría en el proceso del testing.

Para solucionar los problemas anteriores, se propone que la firma, en vez de por trazas, se compute por los pares estado/último evento recibido; que ésta tenga información codificada dentro del valor que se le asigna, y que se utilicen funciones hash para la inserción de los datos dentro de la tabla estática, consiguiendo así un acceso rápido a ellos. Los autores también presentan un estudio realizado sobre eXTP4, un sistema que implementa el protocolo de comunicación TP4 con la metodología de firmado.

En [LCH+02] se hace una extensión del trabajo empezado en [LNS+97] utilizando *Event-driven Extended Finite State Machine*. Para calcular el estado actual del sistema se utiliza información extraída de las variables que aparecen en la especificación. Esta información también se utiliza para comprobar si los comportamientos presentes en el log son o no

erróneos. Para ello se presentan dos algoritmos. El primero es una actualización y mejora del propuesto en [LNS+97], trasladado al nuevo formalismo. El segundo se basa en determinar en cada momento las condiciones sobre los valores de las variables presentes en el log. Este algoritmo, aunque mejora al primero, es incapaz de detectar todos los errores de tipo de cambio de estado. Para superar esta deficiencia, en [ACC+04] se propone una búsqueda de errores no sólo futura, es decir esperando acciones en el log, sino también estudiando lo sucedido. En este nuevo algoritmo se analiza el log hacia el pasado, con el objetivo de reducir aun más la posible configuración inicial de las variables y continuar la comprobación basándose en la especificación. Este nuevo algoritmo fue aplicado con éxito al análisis del protocolo SCP. Por último, en [WSW+07] el estudio y aplicación de algoritmos de testing pasivo al protocolo de comunicaciones TCP, donde además de presentar los resultados con todo detalle, se contempla una posible forma de combinar testing pasivo y activo.

3.3.2. Técnicas basadas en invariantes

En esta sección vamos a revisar el paradigma de testing pasivo con *invariantes*. En este marco, un invariante es una propiedad que se debe cumplir en cualquier estado del sistema. Las técnicas de testing pasivo que se desarrollan dentro de esta tesis se enmarcan en esta categoría. Las contribuciones más importantes dentro de este campo han sido [CGP03, ACN03, BCNZ05].

Una metodología original dentro del testing pasivo se presentó en [CGP03]. En ella no se utiliza el esquema general que hemos estado viendo en otros trabajos de testing pasivo, donde se buscan los fallos de los logs comparándolos con la especificación. La idea novedosa que se introduce es la de trabajar con un conjunto de invariantes que sea capaz de representar las propiedades más importantes a comprobar, y testear los logs contra este conjunto de invariantes. Intuitivamente, un invariante expresa el hecho de que si observamos un cierto comportamiento del sistema, entonces su comportamiento posterior deberá ser conforme a lo que indica el invariante. Los autores proponen un método para extraer invariantes de la especificación. El principal inconveniente de este método es el gran número de invariantes que obtenemos, pudiendo ser incluso infinito. Otro inconveniente que presenta esta primera aproximación al testing pasivo con invariantes se encuentra en la sintaxis de los invariantes, ya que es muy pobre y sólo un conjunto reducido de propiedades se pueden describir con ellos.

En [ACN03] se presenta una extensión, intentando solucionar los problemas comentados. Con respecto a la obtención del conjunto de invariantes, esta nueva metodología permite

que los testeadores puedan definir, sin tener en cuenta a priori la especificación, cualquier conjunto de invariantes. Dado un conjunto de invariantes es importante demostrar que son en efecto correctos con respecto a la especificación. Al efecto, los autores presentaron un algoritmo de corrección. La complejidad del algoritmo es lineal con respecto al número de transiciones de la especificación. A la hora de buscar los errores dentro de los logs, al contrastarlos con los invariantes, los autores implementan una adaptación de los algoritmos clásicos de encaje de patrones [BM77, KMP77]. Además, [ACN03] amplía notablemente la sintaxis de los invariantes: se incluye el símbolo $\star$ para denotar cadenas de acciones de entrada/salida y el símbolo ? para identificar cualquier acción.

La metodología presentada en [ACN03] se extiende en [BCNZ05] con la definición de nuevos tipos de invariantes, la presentación de una herramienta que implementa toda esta metodología y un completo caso de estudio del protocolo WAP. Cabe mencionar que en este protocolo se presenta una situación típica donde no se pueden utilizar las aproximaciones clásicas basadas en testing activo, ya que la determinación de errores se basa en el paso de mensajes entre las diferentes capas de red.

3.4. Runtime verification

En esta sección realizaremos en primer lugar un breve repaso de las *lógicas temporales*, formalismo utilizado para representar las propiedades dentro de las aproximaciones de runtime verification. Después nos centraremos en comentar algunas de las técnicas más representativas.

3.4.1. Lógicas temporales

Las Lógicas Temporales (LT) tienen como propósito representar y razonar acerca de las propiedades de afirmaciones cuyos valores de verdad cambian con el tiempo. A continuación presentamos una pequeña comparación entre ellas [AH93, AH94].

LT proposicionales vs LT primer-orden. En el caso de las proposicionales tenemos que los componentes no temporales del sistema corresponden al cálculo proposicional. Se pueden utilizar tanto variables como constantes, funciones, predicados y cuantificadores, para obtener el mayor poder de expresión del primer orden en el cálculo de predicados. Por otro lado, existen diversos tipos de lógicas temporales de primer orden, dependiendo en gran medida de las *restricciones* del cálculo de predicados.

LT globales vs LT modulares. Si todos los operadores temporales se interpretan en un único universo de discurso, correspondiente a la ejecución del programa, podemos denominar a estas lógicas *globales*.

LT ramificadas vs LT lineales. Dos puntos de vista confluyen sobre la naturaleza del tiempo. El primero considera que para cada instante existe un único momento futuro posible, dando como resultado una estructura de tiempo lineal; frente al segundo punto de vista, que considera que para cada instante existe un conjunto de momentos futuros posibles, dando lugar a una estructura de tiempo arbórea.

LT discretas vs LT continuas. En la mayoría de las lógicas temporales el tiempo se asume discreto, lo que hace que la estructura temporal correspondiente a la ejecución del programa sea una secuencia de estados isomorfa a $\mathbb{N}$, mientras que las lógicas temporales que asumen tiempo continuo se utilizan para dar una mayor expresividad a la hora de representar sistemas reales.

LT de pasado vs LT de futuro. Las primeras lógicas temporales desarrolladas utilizaban modalidades temporales para describir la ocurrencia de eventos en el pasado. Sin embargo, en la mayoría de las lógicas temporales aplicadas al razonamiento, en marcos formales como las álgebra de procesos, se utilizan también modalidades correspondientes al futuro.

3.4.2. Técnicas de runtime verification

La primera técnica que vamos a estudiar es la que se presenta con la herramienta PathExplorer [HR04]. El algoritmo de runtime verification implementado toma como entradas una fórmula lógica temporal de futuro y para la comprobación de la fórmula en estudio genera una adaptación de las máquinas de estados finitos llamada *Binary Transition Tree-Finite State Machine* (BTT-FSM) [dR05]. Los autores señalan que el tamaño de la BTT-FSM generada puede convertirse en un problema crítico del sistema. Por ello, hay que prestar especial atención a las diferentes propuestas de optimización que existen. Este primer algoritmo se recomienda en aquellas situaciones donde el tamaño de las fórmulas a comprobar en relación al tamaño de los logs sea relativamente pequeño.

La implementación en Java de PathExplorer, JPaX, ha sido esencialmente desarrollada y usada en el centro de investigación de la NASA. JPaX se ha aplicado en la producción de muchos programas, tanto para vehículos guiados como para naves aeroespaciales y servicios similares. En particular, se ha utilizado para comprobar el subsistema controlador de vehículos K9 [BDG+04], para el sistema de protección de fallos de vehículos aeroespa-

ciales [BDG+04], y para examinar el comportamiento de dichos vehículos [HR04]. Mientras que el subsistema K9 es una implementación multi-hilo de aproximadamente 8000 líneas de código, JPaX también se ha utilizado en otros entornos con menos líneas de código, o sobre aplicaciones paralelas, donde los resultados experimentales mostraron la utilidad de las tecnologías para aplicaciones críticas [AHB03]. Se ha mostrado que esta técnica es efectiva a la hora de detectar errores en programas grandes, aunque puede producir falsos positivos [AHB03], y puede ser ineficiente en el caso de fallos sutiles [dH05].

Otra metodología de runtime verification a destacar es la de *Monitoring and Checking* (MaC) [SSD+03, KVK+04]. MaC utiliza dos lógicas diferentes para especificar las secuencias de comandos de control y los requisitos de seguridad de los sistemas bajo estudio. Los principales componentes de este marco son los siguientes: en primer lugar se formalizan los requisitos del sistema y se genera un script de monitorización. A continuación, en tiempo de ejecución, se evalúan los eventos a alto nivel que el script anterior se ha encargado de generar. Algunos experimentos con las tecnicas MaC se hicieron en estudios de protocolos [BIS+02] y para la monitorización de agentes [Gor00]. En el primer caso, MaC se utilizó junto con un simulador NS, para comprobar propiedades de un protocolo de enrutado. Los resultados experimentales mostraon la efectividad de esta metodología en la detección de fallos en relación al número de datos que se trataron, pero como desventaja se destaca la gran sobrecarga de recursos que se introducía en el sistema. En el segundo caso, se establece un marco MaC para monitorizar la formación de agentes (nanobots, micro-vehículos aéreos, sistemas micro electro mecánicos, etc.). Los observadores son capaces, en tiempo de ejecución, de reconocer alertas de posibles posiciones erróneas de sus compañeros e inferir la posición actual con respecto a las alertas de otros agentes. Los resultados experimentales obtenidos por JPaX y MaC son convincentes, de manera que son extensamente aplicados a la hora de detección de errores, así como para implementar tolerancia a fallos y sistemas de auto-reparación.

Por último, presentamos las técnicas de *síntesis de programas*. Estas técnicas aprenden aspectos específicos de un sistema en ejecución mediante la mera observación de su comportamiento. En primer lugar, se observa la traza generada por el programa, para producir una síntesis de las propiedades (fase de aprendizaje). A continuación, se compara la ejecución frente a las propiedades que se han inferido. En algunos casos, la fase de aprendizaje no es estática, sino que el programa va aprendiendo y se va adaptando. La detección de propiedades matemáticas sobre variables monitorizadas fue propuesta en [ECGN01] y ha sido implementada en una herramienta llamada Daikon [EPG+07]. Esta técnica requiere la instrumentación del código fuente del programa estudiado para trazar valores interesantes de las variables en

diversos puntos, tales como al comienzo de los procedimientos, al comienzo de los bucles, o al final de los procedimientos. Un problema de esta técnica es su limitada aplicabilidad, ya que sólo manejaba datos de tipo escalar o colecciones. Una solución para aumentar la aplicabilidad en el campo de la programación orientada a objetos consistió en representar los objetos en valores numéricos que además podían agruparse en vectores [EGKN99].

Las técnicas de síntesis de programas se pueden utilizar también para comprobar la compatibilidad de las actualizaciones de los sistemas basados en componentes [ME03, BK08, AGZG08]. En estos casos, la detección de propiedades se ha utilizado para descubrir las pre y post condiciones de los servicios implementados en diferentes módulos. En particular, cuando un componente A de un sistema S es reemplazado por otro elemento T, la compatibilidad de la actualización se establece por *compatibilidad entre pre y post condiciones*. La pre y post condiciones de A se computan mientras el componente es utilizado en S; mientras que la pre y post condiciones de T se computa en la fase de testeo del sistema T.

En [RKS02] se usan estas propiedades para sintetizar el comportamiento de los datos de un *sistema de alimentación de datos*. Los datos se infieren tras observar los resultados, desde un punto de vista del cliente. En particular, en [WLZ07] se utilizan estas técnicas para la extracción de propiedades de servicios online. Otra técnica que merece la pena mencionar es la *Behaviour Capture and Test*, propuesta en [MP05]. Esta metodología verifica propiedades aprendidas en tiempo de ejecución, tanto a nivel de software programado con la metodología de programación orientada a objetos, como de software basado en componentes.

CAPÍTULO 4

Conclusiones y Trabajo Futuro

A menudo encontramos nuestro destino por los caminos que tomamos para evitarlo.

Jean de la Fontaine

El principal objetivo de esta tesis, fue la incorporación de métodos formales en el testing, tanto en el campo de testing activo, mediante la inclusión de modelos probabilísticos, como en el campo del testing pasivo de sistemas, con información temporal. Ambas metodologías, que en un principio parecen difíciles de congeniar, se han conseguido combinar para aprovechar los resultados obtenidos en ambas.

Esta tesis recoge diferentes extensiones de formalismos clásicos. En lo concerniente a la descripción de propiedades probabilísticas, los nuevos modelos pueden expresar la probabilidad que tienen los usuarios del sistema de realizar una determinada acción, mediante el modelo de usuario, o la probabilidad de que ocurra un fallo durante la implementación del sistema, mediante el modelo de implementador. En lo relativo a la inclusión de tiempo, hemos desarrollado diferentes formalismos, tomando como punto de partida las máquinas de estados finitos, para representar sistemas en lo que el tiempo se denota de maneras distintas: bien mediante valores fijos o con variables aleatorias.

En los trabajos que se recogen en esta tesis aparecen diferentes restricciones de aplicabilidad que están presentes en las distintas metodologías de testing; proponiéndose posibles soluciones para la superación de las mismas. Algunas de estas restricciones están exclusivamente relacionadas con los requerimientos temporales de los sistemas, mientras otras son de ámbito general, restringiendo la utilización de algunas técnicas de testing en esos sistemas.

En esta tesis además del carácter teórico de la misma se quiere remarcar su adaptación al *mundo real* (mundo industrial). Para ello hemos incluido estudios de protocolos reales, los cuales nos dan una visión de su adaptabilidad dentro del campo de las telecomunicaciones. La aplicabilidad se ha conseguido con la implementación de nuestra herramienta de análisis de PASive TEsting, que hemos llamado PasTe. Dicha herramienta ha ido evolucionando para incluir a nivel de especificación, automatización, *mutation*, etc. todas las técnicas que se han presentado en esta tesis. Como hito particularmente importante cabe destacar que, como colofón, PasTe ha sido integrada en una herramienta industrial de monitorización, lo que es una muestra relevante de su utilidad y facilidad de uso.

Respecto a las líneas de trabajo futuro, aunque en la tesis hemos abordado muchos de los problemas existentes a la hora de definir propiedades para desarrollar testing pasivo, nos encontramos que, como de costumbre, todavía queda mucho por hacer. Uno de los campos más prometedores para continuar con la investigación que ha dado lugar a esta tesis, se presenta a la hora de utilizar y comparar datos dentro de las trazas y los invariantes. De hecho, se ha empezado a trabajar en esta dirección, definiendo mensajes y sus relaciones, en la última parte de esta tesis. Pero, las variables que aparecen dentro de la especificación,

la corrección de nuevos invariantes frente a esas variables y la obtención automática de éstos, representa un campo que, por el momento, no ha podido ser explorado con la misma profundidad con la que hemos tratado el campo de las extensiones temporales. Por todo ello, consideramos que la creación de un marco de testing pasivo que permita expresar y analizar propiedades sobre datos, colecciones de valores, y topología de los mismos, podría ser de gran utilidad, si tenemos en cuenta que dichos datos son un aspecto crítico en el comportamiento de los sistemas reales. Consideramos que esta línea de investigación es lo suficientemente amplia para que dé lugar al menos a alguna nueva tesis doctoral.

BIBLIOGRAFÍA

[AAD79] J.M. Ayache, P. Azema, and M. Diaz. Observer: A concept for on-line detection of control errors in concurrent systems. In *9th Symposium on Fault-Tolerant Computing*, 1979.

[ABL05] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th Int. Conf. on Software Engineering, ICSE'05*, pages 402–411. ACM Press, 2005.

[Abr96] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ACC+04] B. Alcalde, A.R. Cavalli, D. Chen, D. Khuu, and D. Lee. Network protocol system passive testing for fault management: A backward checking approach. In *FORTE 2004, LNCS 3235*, pages 150–166. Springer, 2004.

[ACN03] J.A. Arnedo, A. Cavalli, and M. Núñez. Fast testing of critical properties through passive testing. In *15th Int. Conf. on Testing Communicating Systems, TestCom'03, LNCS 2644*, pages 295–310. Springer, 2003.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148:1–70, 1999.

[AGZG08] R. Abreu, A. González, P. Zoeteweij, and V. Gemund. Automatic software fault localization using generic program invariants. In *23rd ACM Symposium on Applied Computing, SAC'08*, pages 712–717. ACM Press, 2008.

[AH93] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[AH94] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, 1994.

[AHB03] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[Ale07] S. Alekseev. Java debugging laboratory for automatic generation and analysis of trace data. In *25th IASTED Conf. on Software Engineering, SE'07*, pages 177–182. ACTA Press, 2007.

[ALR08] C. Andrés, L. Llana, and I. Rodríguez. Formally comparing user and implementer model-based testing methods. In *4th Workshop on Advances in Model Based Testing, A-MOST'08*, pages 1–10. IEEE Computer Society Press, 2008.

[ALR09] C. Andrés, L. Llana, and I. Rodríguez. Formally transforming user-model testing problems into implementer-model testing problems and viceversa. *Journal of Logic and Algebraic Programming*, 78(6):425–453, 2009.

[AMC+09] C. Andrés, S. Maag, A. Cavalli, M.G. Merayo, and M. Núñez. Analysis of the OLSR protocol by using formal passive testing. In *16th Asia-Pacific Software Engineering, APSEC'09*, pages 152–159. IEEE Computer Society Press, 2009.

[AMM09] C. Andrés, M.G. Merayo, and C. Molinero. Advantages of mutation in passive testing: An empirical study. In *4th Workshop on Mutation Analysis, Mutation'09*, pages 230–239. IEEE Computer Society Press, 2009.

[AMN08] C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of timed systems. In *6th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'08, LNCS 5311*, pages 418–427. Springer, 2008.

[AMN09a] C. Andrés, M.G. Merayo, and M. Núñez. Formal correctness of a passive testing approach for timed systems. In *5th Workshop on Advances in Model Based Testing, A-MOST'09*, pages 67–76. IEEE Computer Society Press, 2009.

[AMN09b] C. Andrés, M.G. Merayo, and M. Núñez. Formal passive testing of timed systems: A case study with the Stream Control Transmission Protocol. In *7th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM'09*, pages 73–82. IEEE Computer Society Press, 2009.

[AMN09c] C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of stochastic timed systems. In *2nd Int. Conf. on Software Testing, Verification, and Validation, ICST'09*, pages 71–80. IEEE Computer Society Press, 2009.

[AMN09d] C. Andrés, M.G. Merayo, and M.Ñúñez. Supporting the extraction of timed properties for passive testing by using probabilistic user models. In *9th Int. Conf. on Quality Software, QSIC'09*, pages 145–154. IEEE Computer Society Press, 2009.

[AMN09e] C. Andrés, M.G. Merayo, and M. Núñez. Using a mining frequency patterns model to automate passive testing of real-time systems. In *21st Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'09*, pages 426–431. Knowledge Systems Institute, 2009.

[And08] C. Andrés. Two quantitative extensions to perform formal testing of systems. Master's thesis, Universidad Complutense de Madrid, 2008.

[AW96] S.P. Allen and M.R. Woodward. Assessing the quality of specification-based testing. In *3rd Int. Conf. on Achieving Quality in Software, AQuIS'96*, pages 341–354. Chapman & Hall, 1996.

[BCNZ05] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: Application to the WAP. *Computer Networks*, 48(2):247–266, 2005.

[BDG+04] G.P. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M.R. Lowry, C.S. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.

[BDS+07] A. Benharref, R. Dssouli, M.A. Serhani, A. En-Nouaary, and R. Glitho. New approach for EFSM-based passive testing of web services. In *Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems,*

TestCom'07, and 7th Int. Workshop on Formal Approaches to Software Testing, FATES'07, LNCS 4581, pages 13–27. Springer, 2007.

[BIS+02] K. Bhargavan, C. Gunter L. Insup, O. Sokolsky, M. Kim, D. Obradovic, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, 2002.

[BK08] K.N. Biyani and S.S. Kulkarni. Assurance of dynamic adaptation in distributed systems. *Journal of Parallel and Distributed Computing*, 68(8):1097–1112, 2008.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BM99] L. Bottaci and E.S. Mresa. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[BM04] M. Bidoit and P. Mosses, editors. *CASL Reference User Manual: Introduction to using the Common Algebraic Specification Language (LNCS 2900)*. Springer, 2004.

[BP94] G. von Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *ACM Int. Symposium on Software Testing and Analysis, ISSTA'94*, pages 109–123. ACM Press, 1994.

[BPBM97] G. von Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga. Automating the process of test derivation from SDL specifications. In *8th Int. SDL Forum, SDL'97*, pages 261–276. Elsevier, 1997.

[BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North Holland, 2001.

[BS03] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.

[BT01] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 187–195. Springer, 2001.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Computer Science 18. Cambridge University Press, 1990.

[BY05] L. Baresi and M. Young. Toward translating design constraints to run-time assertions. *Theoretical Computer Science*, 116:73–84, 2005.

[CCI88] SDL: Specification and design language, 1988. CCITT Recommendations Z.101-Z.104, *Blue Book Series*. Consultative Committe for International Telegraph and Telephone.

[CFP96] A. Cavalli, J.P. Favreau, and M. Phalippou. Standardization of formal methods in conformance testing of communication protocols. *Computer Networks and ISDN Systems*, 29:3–14, 1996.

[CGP03] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837–852, 2003.

[Cho78] T.S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[Cho86] N. Choquet. Test data generation using a prolog with constraints. In *Workshop on Software Testing*, pages 132–141. IEEE Computer Society Press, 1986.

[CHR82] L.A. Clarke, J. Hassell, and D.J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8(4):380–390, 1982.

[Chr90] I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In *1st Int. Conf. on Concurrency Theory, CONCUR'90, LNCS 458*, pages 126–140. Springer, 1990.

[CS94] D.A. Carrington and P.A: Stocks. A tale of two paradigms: Formal methods and software testing. In *Z User Workshop*, pages 51–68. Springer, Workshops in Computing, 1994.

[CSV07] L. Cheung, M. Stoelinga, and F. Vaandrager. A testing scenario for probabilistic processes. *Journal of the ACM*, 54(6):Article 29, 2007.

[CTC01] H.Y. Chen, T.H. Tse, and T.Y. Chen. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.

[CTCC98] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.

[DB99] J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9(1):27–50, 1999.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model based specifications. In *1st. Int. Symposium of Formal Methods Europe, FME'96, LNCS 670*, pages 268–284. Springer, 1993.

[DF94] R.-K. Doong and P.G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[DGR04] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[dH05] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[DJC94] M. Diaz, G. Juanole, and J.P. Courtiat. Observer-a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, 1994.

[dNH84] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[dR05] M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *17th Int. Conf. on Computer Aided Verification, CAV'05, LNCS 3576*, pages 364–378. Springer, 2005.

[DS03] D. Drusinsky and M.-T. Shing. Monitoring temporal logic specifications combined with time series constraints. *Journal of Universal Computer Science*, 9(11):1261–1276, 2003.

[DU04] A.Y. Duale and M.Ü. Uyar. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Transactions on Computers*, 53(5):614–627, 2004.

[ECGN01] M.D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[EGKN99] M.D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, 1999.

[Eil74] S. Eilenberg. *Automata, languages and machines*, volume A. Academic Press, 1974.

[EPG+07] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[FRS94] J. Farnam, R. Ragunathan, and C.V.R. Sitaram. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, 1994.

[Gau95] M.-C. Gaudel. Testing can be formal, too! In *6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, pages 82–96. Springer, 1995.

[Gau01] M.-C. Gaudel. Testing from formal specifications, a generic approach. In *6th Int. Conf. Ada-Europe, LNCS 2043*, pages 35–48. Springer, 2001.

[GG75] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[GJ98] M.-C. Gaudel and P.R. James. Testing algebraic data types and processes: A unifying theory. *Formal Aspects of Computing*, 10(5-6):436–451, 1998.

[GLNP97] C. Gregorio, L. Llana, M. Núñez, and P. Palao. Testing semantics for a probabilistic-timed process algebra. In *4th International AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software, LNCS 1231*, pages 353–367. Springer, 1997.

[GMH81] J.D. Gannon, P.R. McMullin, and R.G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[Gon70] G. Gonenc. A method for the design of fault detection experiments. *IEEE Transactions on Computers*, 19:551–558, 1970.

[Gor00] D. F. Gordon. Asimovian adaptive agents. *Artificial Intelligence Research*, 13:95–153, 2000.

[GT79] J.A. Goguen and J.J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic specifications. In *IEEE Conf. on Specifications of Reliable Software*, pages 170–189. IEEE Computer Society Press, 1979.

[Hal89] P.A.V. Hall. Towards testing with respect to formal specification. In *2nd IEE/BCS Conference on Software Engineering*, pages 159–163, 1989.

[Har87] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HBB+09] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal methods to support testing. *ACM Computing Surveys*, 41(2), 2009.

[Hen64] F.C. Hennie. Fault-detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[Hen85] M. Hennessy. Acceptance trees. *Journal of the ACM*, 32(4):896–928, 1985.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HH00] R.M. Hierons and M. Harman. Testing comformance to a quasi-non-determinstic stream X-machine. *Formal Aspects of Computing*, 12(6):423–442, 2000.

[HH04] R.M. Hierons and M. Harman. Testing conformance of a deterministic implementation to a non-deterministic stream X-machine. *Theoretical Computer Science*, 323(1–3):191–233, 2004.

[HI98] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer, 1998.

[Hie97] R.M. Hierons. Testing from a Z specification. *Software Testing, Verification and Reliability*, 7(1):19–33, 1997.

[HM09] R.M. Hierons and M.G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software (in press)*, 2009.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol88] M. Holcombe. X–machines as a basis for dynamic system specification. *Software Engineering Journal*, 3(2):69–76, 1988.

[How82] W.E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379, 1982.

[HR95] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

[HR04] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[HSS01] R.M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using statecharts and Z. *Information and Software Technology*, 43(2):137–149, 2001.

[HU02] R.M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117, 2002.

[IH97] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63(3-4):159–178, 1997.

[IH00] F. Ipate and M. Holcombe. Generating test sets from non-deterministic stream X-machines. *Formal Aspects of Computing*, 12(6):443–458, 2000.

[Ipa04] F. Ipate. Complete deterministic stream X-machine testing. *Formal Aspects of Computing*, 16(4):374–386, 2004.

[IT97] ITU-T. *Recommendation Z.500 Framework on formal methods in conformance testing*. International Telecommunications Union, Geneva, Switzerland, 1997.

[ITU92] ITU. Recommendation Z.100: CCITT Specification and Description Language (SDL), 1992.

[Jal83] P. Jalote. Specification and testing of abstract data types. In *7th Int. Computer Software and Applications Conference, COMPSAC'83*, pages 508–511. IEEE Computer Society Press, 1983.

[JC88] P. Jalote and M.G. Caballero. Automated test case generation for data abstraction. In *12th Int. Computer Software and Applications Conference, COMPSAC'88*, pages 205–210. IEEE Computer Society Press, 1988.

[Jon91] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, 2nd edition, 1991.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated test generation from SDL specifications. In *9th Int. SDL Forum, SDL'99*, pages 135–152. Elsevier, 1999.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[Koh78] Z. Kohavi. *Switching and Finite State Automata Theory.* McGraw-Hill, 1978.

[KVK+04] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A runtime assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[Lay92] G.T. Laycock. *The Theory and Practice of Specification Based Software Testing.* PhD thesis, University of Sheffield, 1992.

[LCH+02] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In *10th IEEE Int. Conf. on Network Protocols, ICNP'02*, pages 122–131. IEEE Computer Society Press, 2002.

[LCH+06] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu, and X. Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, 14:424–437, 2006.

[LdF99] L. Llana and D. de Frutos. Relating may and must testing semantics for discrete timed process algebras. In *5th Asian Computing Science Conference, ASIAN'99, LNCS 1742*, pages 74–86. Springer, 1999.

[Led91] G. Leduc. Conformance relation, associated equivalence, and minimum canonical tester in LOTOS. In *11th WG6.1 Int. Conf. on Protocol Specification, Testing, and Verification, PSTV'91*, pages 249–264. North Holland, 1991.

[LMK07] G. Lee, A.K. Mok, and P. Konana. Monitoring of timing constraints with confidence threshold requirements. *IEEE Transactions on Computers*, 56(7):977–991, 2007.

[LNR06a] L.F. Llana, M. Núñez, and I. Rodríguez. Customized testing for probabilistic systems. In *18th Int. Conf. on Testing Communicating Systems, TestCom'06, LNCS 3964*, pages 87–102. Springer, 2006.

[LNR06b] L.F. Llana, M. Núñez, and I. Rodríguez. Derivation of a suitable finite test suite for customized probabilistic systems. In *26th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'06, LNCS 4229*, pages 467–483. Springer, 2006.

[LNR06c] N. López, M. Núñez, and I. Rodríguez. Specification, testing and implementation relations for symbolic-probabilistic systems. *Theoretical Computer Science*, 353(1–3):228–248, 2006.

[LNS+97] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *5th IEEE Int. Conf. on Network Protocols, ICNP'97*, pages 113–122. IEEE Computer Society Press, 1997.

[LOT88] LOTOS. A formal description technique based on the temporal ordering of observational behaviour. IS 8807, TC97/SC21, 1988.

[LPU02] B. Legeard, F. Peureux, and M. Utting. A comparison of the BTT and TTF test-generation methods. In *2nd Int. Conf. of B and Z Users, LNCS 2272*, pages 309–329. Springer, 2002.

[LS89] K. Larsen and A. Skou. Bisimulation through probabilistic testing. In *16th ACM Simposium on Principles of Programming Languages, POPL'89*, pages 344–352. ACM Press, 1989.

[LS94] S. Lee and K. G. Shin. Probabilistic diagnosis of multiprocessor systems. *ACM Computer Surveys*, 26(1):121–139, 1994.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[MA00] R.E. Miller and K.A. Arisha. On fault location in networks by passive testing. In *19th IEEE Int. Performance, Computing, and Communications Conference, IPCCC'00*, pages 281–287. IEEE Computer Society Press, 2000.

[MA01] R.E. Miller and K.A. Arisha. Fault identification in networks by passive testing. In *34th Simulation Symposium, SS'01*, pages 277–284. IEEE Computer Society Press, 2001.

[ME03] S. McCamant and M.D. Ernst. Predicting problems caused by component upgrades. In *9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of software engineering, ESEC/FSE-11*, pages 287–296. ACM Press, 2003.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems (LNCS 92)*. Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil98] R.E. Miller. Passive testing of networks using a CFSM specification. In *IEEE Int. Performance Computing and Communications Conference*, pages 111–116. IEEE Computer Society Press, 1998.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[Moo56] E.P. Moore. Gedanken experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[Mos04] P. Mosses, editor. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language (LNCS 2960)*. Springer, 2004.

[MP05] L. Mariani and M. Pezzè. A technique for verifying component-based software. *Electronic Notes in Theoretical Computer Science*, 116:17–30, 2005.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I & II. *Information and Computation*, 100:1–77, 1992.

[NdF95] M. Núñez and D. de Frutos. Testing semantics for probabilistic LOTOS. In *8th IFIP WG6.1 Int. Conf. on Formal Description Techniques, FORTE'95*, pages 365–380. Chapman & Hall, 1995.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines. In *IEEE Fault Tolerant Computer Systems*, pages 238–243. IEEE Computer Society Press, 1981.

[Núñ03] M. Núñez. Algebraic theory of probabilistic processes. *Journal of Logic and Algebraic Programming*, 56(1–2):117–177, 2003.

[NVN98] G. Noubir, K. Vijayananda, and H. J. Nussbaumer. Signature-based method for run-time fault detection in communication protocols. *Computer Communications*, 21(5):405–421, 1998.

[Off94] J. Offutt. A practical system for mutation testing: Help for the common programmer. In *7th International Test Conference, ITC'94*, pages 824–830. IEEE Computer Society Press, 1994.

[Paz71] A. Paz. *Introduction to Probabilistic Automata*. Academic Press, 1971.

[PBG04] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering*, 30(1):29–42, 2004.

[Pha94] M. Phalippou. Executable testers. In *6th IFIP Int. Workshop on Protocol Test Systems, IWPTS'93*, pages 35–50. North-Holland, 1994.

[PYB96] A. Petrenko, N. Yevtushenko, and G. von Bochmann. Fault models for testing in context. In *Formal Description Techniques for Distributed Systems and Communication Protocols (IX), and Protocol Specification, Testing, and Verification (XVI)*, pages 163–178. Chapman & Hall, 1996.

[Rab63] M.O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

[RKS02] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *24th IEEE Int. Conf. on Software Engineering, ICSE'02*, pages 302–312. ACM Press, 2002.

[RMN08] I. Rodríguez, M.G. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.

[SCS97] H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. In *1st IEEE Int. Conf. on Formal Engineering Methods, ICFEM'97*, pages 81–90. IEEE Computer Society Press, 1997.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[SEK+98] M. Schmitt, A. Ek, B. Koch, J. Grabowski, and D. Hogrefe. Autolink - putting SDL-based test generation into practice. In *11th IFIP Workshop on Testing of Communicating Systems, IWTCS'98*, pages 227–244. Kluwer Academic Publishers, 1998.

[SF07] Y. Song and B. D. Fleisch. Utilizing binary rewriting for improving end-host security. *IEEE Transactions on Parallel and Distributed Systems*, 18(12):1687–1699, 2007.

[SM93] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, 26(3), 1993.

[SMIM90] F. Sato, J. Munemori, T. Ideguchi, and T. Mizuno. Sequence generation tool for communication systems. In *2nd WG6.1 Int. Conf. on Formal Description Techniques, FORTE'89*, pages 1–5. North-Holland, 1990.

[Sos92] R. Sosič. Dynascope: a tool for program directing. *ACM SIGPLAN Notice*, 27(7):12–21, 1992.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition, 1992.

[SSD+03] U. Sammapun, R. Sharykin, M. DeLap, M. Kim, and S. Zdancewic. Formalizing Java-MaC. *Theoretical Computer Science*, 89(2):171–190, 2003.

[TC99] M. Tabourier and A. Cavalli. Passive testing and application to the GSM-MAP protocol. *Information and Software Technology*, 41(11-12):813–821, 1999.

[TJ98] K. Templer and C. Jeffery. A configurable automatic instrumentation tool for ANSI C. In *13th IEEE Int. Conf. on Automated software engineering, ASE '98*, pages 249–258. IEEE Computer Society Press, 1998.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.

[USW00] H. Ural, K. Saleh, and A. Williams. Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications*, 23:609–627, 2000.

[UWZ97] H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.

[UX07] H. Ural and Z. Xu. An EFSM-based passive fault detection approach. In *Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'07, and 7th Int. Workshop on Formal Approaches to Software Testing, FATES'07, LNCS 4581*, pages 335–350. Springer, 2007.

[UXZ07] H. Ural, Z. Xu, and F. Zhang. An improved approach to passive testing of FSM-based systems. In *2nd Int. Workshop on Automation of Software Test, AST'07*, page 6. IEEE Computer Society Press, 2007.

[Vas73] M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 4:653–665, 1973.

[WC80] L.J. White and E.I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, 1980.

[WLZ07] C. Wang, J. Lu, and G. Zhang. Mining key information of web pages: A method and its application. *Expert Systems with Applications*, 33(2):425–433, 2007.

[Wol99] M. Wolczko. Using a tracing javaTM virtual machine to gather data on the behavior of java programs. Technical Report SML-98-0154, Sun Microsystems, 1999.

[Woo93] M.R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *IEE/BCS Software Engineering Journal*, 8(4):211–224, 1993.

[WS93] C. Wang and M. Schwartz. Fault detection with multiple observers. *IEEE/ACM Transactions on Networking*, 1(1):48–55, 1993.

[WSW+07] W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley. Passive online rogue access point detection using sequential hypothesis testing with TCP ACK-pairs. In *7th ACM SIGCOMM Internet Measurement Conference, IMC '07*, pages 365–378. ACM Press, 2007.

[WZY01] J. Wu, Y. Zhao, and X. Yin. From active to passive: Progress in testing of internet routing protocols. In *21st IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'01*, pages 101–116. Kluwer Academic Publishers, 2001.

[YWS+89] S.W. Yeh, C. Wu, H.D. Sheng, C.K. Hung, and R.C. Lee. Expert system based automatic network fault management system. In *13th Annual Int. Computer Software and Applications Conference, COMPSAC'89*, pages 767–774. IEEE Computer Society Press, 1989.

[ZYW03] Y. Zhao, X. Yin, and J. Wu. Problems in the information dissemination of the internet routing. *Journal of Computer Science and Technology*, 18(2):139–152, 2003.